UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

TRABAJO FIN DE GRADO

SISTEMA DE MONITORIZACIÓN PARA CONVERTIDORES DE POTENCIA

Antonio Ortega Fernández

Tutor: Alberto Sánchez González

Ponente: Ángel de Castro Martín

ENERO 2015

SISTEMA DE MONITORIZACIÓN PARA CONVERTIDORES DE POTENCIA

AUTOR: Antonio Ortega Fernández

TUTOR: Alberto Sánchez González

Trabajo realizado en el grupo

Human Computer Technology Laboratory

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Enero 2015

AGRADECIMIENTOS

En primer lugar me gustaría empezar dando las gracias a mi magnífico tutor, Alberto Sánchez, por toda su dedicación, ayuda, esfuerzo, por haberme enseñado tantísimas cosas a lo largo de estos meses, y por dedicarme tanto de su tiempo, sin él este trabajo no hubiese sido posible. No puedo olvidar tampoco a mi ponente, Ángel de Castro, por haberme prestado su ayuda en las distintas gestiones del TFG.

Gracias también a mis padres, por su confianza y apoyo incondicional durante todos estos años; y en especial a mi hermana María, por creer siempre en mí.

Gracias a todos los compañer@s y amig@s de universidad que me han acompañado durante estos tortuosos años de estudio; en especial a Álvaro, Lucas, Álex, Carlos, Raquel, Miriam, Miguel el tránsfuga y Paco, por los momentos compartidos tanto dentro de la universidad, como fuera. También gracias a todos mis compañeros del "Wolf Pack", por las experiencias vividas durante mi inolvidable estancia en Suecia.

Mención especial también merecen mis amigos de Almería de toda la vida, Antonio el grande, Jero, David, Dark, Mj, Nerea, Vargas, Ruso, Menchu, Javi y Rubén, a los cuales aprecio un montón, por hacer que cada vez que vuelvo a Almería es como si nunca me hubiese ido.

Y en general, gracias a todas las personas que hayan podido contribuir a hacer posible este Trabajo Fin de Grado; tanto a los que siguen, como a los que se fueron.

Antonio Ortega Fernández.

Enero 2015.

RESUMEN

Los convertidores de potencia conmutados han ido instalándose en la industria durante las últimas décadas. Estas fuentes que basan su funcionamiento en la conmutación de un interruptor, consiguen una mayor eficiencia energética, siendo ésta una gran ventaja frente a las fuentes lineales.

El control de las fuentes conmutadas inicialmente ha sido de tipo analógico, pero el avance de los dispositivos digitales y la reducción de sus costes ha promovido la aparición de fuentes conmutadas controladas con dispositivos digitales, bien sean con circuitos integrados específicos, DSPs (*Digital Signal Processing*), FPGAs (*Field Programmable Gate Array*) y en incluso microcontroladores.

Los controladores digitales presentan un gran número de ventajas como pueden ser la reprogramabilidad y la posibilidad de realizar algoritmos de control más complejos. Otra gran atributo es la facilidad de ofrecer una interfaz entre el propio controlador de la fuente y un dispositivo externo, como puede ser un ordenador. De esa forma, el operario o usuario de la fuente podría visualizar y modificar el comportamiento de dicha fuente gracias a los datos recibidos.

Este Trabajo Fin de Grado presenta el desarrollo de una interfaz que permite la monitorización de un convertidor elevador, controlado digitalmente mediante una FPGA, a través de un ordenador. Se mostrará el desarrollo de todos los módulos necesarios de código para la comunicación (tanto en la FPGA como en el ordenador), y todos los elementos *hardware* necesarios, así como la definición del protocolo de comunicación.

El protocolo de comunicación se basará en una transmisión serie de tipo UART, ya que ofrece gran simplicidad. Para que la monitorización pueda realizarse desde cualquier ordenador, también se va a mostrar el diseño de un PCB (*Printed Circuit Board*) para la conversión UART-USB, por lo que cualquier ordenador con puerto USB podrá utilizar el sistema de monitorización propuesto.

En particular, este Trabajo Fin de Grado consistirá en la monitorización de las tensiones de entrada y salida, así como la corriente de entrada de un convertidor elevador controlado para corrección de factor de potencia. Estas tres magnitudes físicas se enviarán en tiempo real a través del protocolo de comunicación citado, y la aplicación gráfica permitirá representar dicho información, así como configurar parámetros de la visualización, como puede ser longitud de la visualización, configuración de *triggers*, etc.

PALABRAS CLAVE

Electrónica de potencia, monitorización, Universal Asynchronous Receiver/Transmitter, Field Programmable Gate Array, C sharp.

ABSTRACT

Switching-mode power converters have had a significant growth over the last decade. These converters are based on the *switching power pole* principle, and are able to reach high energy efficiency ratings, being this their main attractive.

Analog regulators were firstly used to control these switching-mode power converters for many decades, but digital devices' lately progression has turned it around and digital control has become one of the main characters when dealing with switching-mode power converters regulation. These power converters can be developed on other specific integrated circuits such as DSPs (*Digital Signal Processing*), FPGAs (*Field Programmable Gate Array*) or even micro controllers, mainly due to its cost reduction and higher efficiency previously cited.

Digital controllers present many advantages over analog ones, for instance, reprogrammability, or the capability of implementing more complex control algorithms. Offering an interface between the power supply controller and an external device, like a personal computer, is another good attribute of digital controllers. In this way, a user could visualize and monitor the behavior of this power supply thanks to the digitalized data.

This degree project presents the development of a user interface that allows the motorization of a step-up converter, also known as *boost converter*, by means of a FPGA and throughout a personal computer. The development of the different modules needed for the communication protocol (both in the FPGA and the personal computer) that integrate this project, as well as all hardware elements and the definition of this communication protocol, will be explained through this degree project report.

The communication protocol will be based on the UART serial transmission protocol, due to its simplicity. In order to being able to monitor the data on any computer, a PCB (*Printed Circuit Board*) will be designed, allowing the UART-USB conversion and thus its connection to any computer with an USB port.

In particular, this degree project will consist in the monitoring of both the input and output voltage, as well as the input current of the step-up convertor for the power factor corrector. These three physical magnitudes will be sent on real time to the user interface, though the communication protocol previously cited. This user interface will receive the data, and will set up the display parameters, like the length of the display, or a possible *trigger* configuration.

KEYWORDS

Power electronics, monitoring, Universal Asynchronous Receiver/Transmitter, Field Programmable Gate Array, C sharp.

ÍNDICE GENERAL

INDICE DE FIGURAS

INDICE DE TABLAS

1. INTRODUCCIÓN

La electrónica de potencia provee una interfaz entre una fuente determinada (*source* en la figura 1) y una carga (*load*), que hace posible la transferencia de energía entre ambas, transformando los valores de tensión y corriente de la entrada a la salida, según sea necesario. Dicha fuente y carga pueden, y frecuentemente así ocurre, diferir en frecuencia, amplitudes de voltajes, y fases. Esta conversión de corrientes y voltajes debería ser llevada a cabo con la mayor eficiencia energética y la mayor densidad de potencia posible [1].

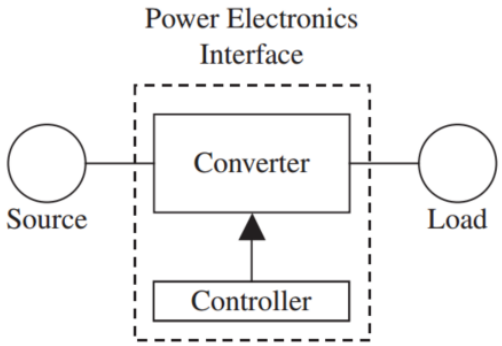

Fig. 1: Esquema básico en lazo cerrado de un convertidor de potencia. [2]

Un convertidor es por tanto la interfaz mencionada que existe entre una fuente de energía y una carga, y que permite variar las características de la corriente o tensión recibidas en base a unos requerimientos determinados.

Desde hace décadas, se ha venido trabajando principalmente con convertidores de potencia conmutados (basados en interruptores), ya que con este tipo de convertidores se consigue mejorar la eficiencia de conversión de manera significativa.

Existen numerosas topologías de convertidores conmutados, entre las que destacan tres tipos de topologías sencillas:

- Convertidor elevador (*Boost*): con el que se obtiene una tensión de salida mayor a la tensión de entrada.

- Convertidor reductor (*Buck*): con el que se obtiene una tensión de salida menor que la tensión de entrada.

- Convertidor elevador-reductor (*Buck-Boost*): es un convertidor con el que se pueden conseguir tensiones de salida mayores a la de entrada (actuando así como un convertidor elevador), o menores a la de entrada (actuando por tanto como un convertidor reductor).

Los convertidores de potencia conmutados, como se comentaba, pueden alcanzar eficiencias hasta del 95% en sistemas reales [2], y pueden reducir a su vez el gasto energético del sistema. Debido a esta reducción en la potencia disipada, son por tanto una mejora sustancial con respecto a otros convertidores de potencia, donde las eficiencias obtenidas son mucho menores.

Ahorrar energía en cualquier proceso electrónico (reducir la potencia disipada en el sistema), es algo de primera necesidad en los sistemas actuales, y es por tanto que en las últimas décadas se ha venido investigando cómo optimizar esta conversión de energía. La reducción de energía disipada permite, no sólo ahorro económico, sino producir menor coste medioambiental y reducir la generación de calor en el convertidor de potencia, algo que es crítico en muchas aplicaciones.

Como se planteó en el resumen, los convertidores de potencia pueden ser controlados analógica o digitalmente, aunque hoy día lo más extendido es principalmente el control digital, debido a que presenta numerosas ventajas frente al control analógico [3], como son: la reprogramación de los dispositivos, la reducción del tiempo de diseño, la fiabilidad y la integración de los mismos.

En el caso de disponer de un control digital, es necesario digitalizar las señales a controlar (normalmente tensiones y corrientes) en lazo cerrado, mediante el uso de conversores analógico digital (*ADC*). Aprovechando la citada digitalización, este Trabajo Fin de Grado implementa un sistema de monitorización sencillo y barato para que un dispositivo externo pueda conocer qué está ocurriendo en el convertidor de potencia, así como en el regulador.

1.1. Motivación y objetivos

Los convertidores de potencia conmutados anteriormente explicados, están ampliamente extendidos y son utilizados en infinidad de sistemas electrónicos, ya sean a nivel doméstico, militar o industrial.

Es por tanto que juegan un papel fundamental en el correcto funcionamiento del sistema electrónico completo del que forman parte. Debido a este papel fundamental, es primordial en un sistema electrónico de altas prestaciones, que los convertidores de potencia conmutados utilizados, garanticen la mayor eficacia posible y se ajusten de la manera más precisa posible a los requerimientos especificados.

Partiendo de esta motivación, los objetivos principales de este Trabajo Fin de Grado son tres:

- Realizar un sistema de monitorización de los datos digitalizados en la regulación del convertidor de potencia. Para ello, se debe añadir un módulo de comunicación con el exterior que no interfiera con el controlador del convertidor de potencia.
- Desarrollar una interfaz gráfica de usuario para poder rescatar y analizar estos datos de control.

- Construir un conversor UART/USB, ya que la interfaz USB está muy extendida. Especialmente, la utilización del protocolo USB permitirá que los ordenadores actuales puedan utilizar el sistema de monitorización, ya que el puerto COM-RS2323 está actualmente en desuso en la informática de consumo.

1.2. Estructura de la memoria

Esta memoria se estructura de la siguiente manera:

- En el primer capítulo, se ha realizado una breve introducción del TFG, así como de su motivación y los objetivos que se persiguen con el mismo.

- En el segundo capítulo se expondrá el estado del arte de este TFG, discutiendo mayoritariamente acerca de los convertidores elevadores *boost*, así como de la corrección del factor de potencia.

- En el tercer capítulo se muestra una visión global del proyecto implementado, para exponer de manera concisa el objetivo de dicho proyecto. Se describirán brevemente cada uno de los módulos que componen el TFG, pero sin entrar en mayor detalle puesto que la información más pormenorizada de cada apartado se proveerá en las secciones correspondientes a cada módulo.

- En el cuarto capítulo se explicará y desarrollará el módulo de comunicación y todos los dispositivos que lo conforman: conversores analógico digital, FPGA, UART y DCM. Se añadirán esquemas, tablas, infografía, fragmentos del código desarrollado… en función a las necesidades de cada sub-apartado.

- En el quinto capítulo tiene lugar la explicación del módulo de construcción, en el que se detallarán los pasos que se han seguido para construir el circuito impreso utilizado en este proyecto, así como las características del microchip FT2323R empleado.

- En el sexto capítulo se formulará el módulo de simulación, y se detallarán las secciones de la aplicación desarrollada en lenguaje C# utilizando *software* de Microsoft.

- En el séptimo capítulo se expondrán los resultados obtenidos en la interfaz gráfica de usuario al integrar los distintos módulos dentro del mismo sistema.

- Por último, se recogerán las conclusiones obtenidas tras el desarrollo de este proyecto, así como bibliografía, glosarios y anexos.

2. ESTADO DEL ARTE

2.1. Factor de potencia

La corrección del factor de potencia (*Power Factor Correction, PFC,* en inglés) es algo que lleva estudiándose y aplicándose desde hace décadas. La corrección del factor de potencia consigue rectificar la tensión alterna de entrada, a la vez que se regula la tensión media de salida y la corriente de entrada, o lo que es lo mismo, consigue mejorar su factor de potencia [4].

El factor de potencia, f.d.p., de un convertidor de potencia, se define como la relación entra la potencia activa (la potencia consumida por los dispositivos eléctricos), y la potencia aparente (la suma de la potencia activa y la potencia reactiva), y es por tanto una medida de la capacidad de una carga para absorber potencia activa [5]. Si se consigue hacer que la potencia de la carga parezca ser puramente resistiva (que la potencia aparente sea igual a la potencia real), se consigue también que el voltaje y la corriente estén en fase, que la potencia reactiva consumida sea 0, y que por tanto esto favorezca una mayor eficiencia en el circuito.

Consiguiendo un factor de potencia elevado, se consigue optimizar técnica y económicamente una instalación: se logra disminuir la sección de los cables utilizados, disminuir las pérdidas en las líneas, reducir la caída de tensión y aumentar la potencia disponible. De hecho, hay normativas como la normativa IEC 61000-3-2, publicada por la *International Electrotechnical Comission* (IEC), que obliga a las fuentes de alimentación a cumplir factores de potencia mínimos [5].

La fórmula del factor de potencia viene dada por:

$$f.d.p. = \frac{p_{aparente}}{\sqrt{p_{aparente}^2 + p_{reactiva}^2}}$$

El factor de potencia, que tomará un valor entre 0 y 1, indica si hay desfase o no entra la corriente y el voltaje [6]. A menor factor de potencia, mayor desfase habrá. Idealmente, esta relación es igual a 1, indicando máxima eficiencia en la conversión, produciéndose cuando la carga cumple la ley de Ohm, es decir, si la carga se comporta como una resistencia. Esto produce que la tensión y la corriente de entrada sean proporcionales y el contenido armónico sea nulo [4]. La conversión clásica AC/DC mediante un puente de diodos obtiene factores de potencia muy bajos, por lo que es necesario aplicar técnicas de corrección de factor de potencia o PFC.

Las técnicas PFC permiten reducir el contenido armónico que se produce al rectificar la tensión de entrada mediante una fuente conmutada, emulando una carga resistiva. Dichas técnicas se pueden aplicar a diferentes tipos de convertidores, para el caso de este TFG se va a utilizar un convertidor elevador *boost*.

2.2. Convertidor elevador *Boost*

El convertidor elevador tipo, o *boost converter* en inglés, es un conversor de DC/DC, o AC/DC, dependiendo del sistema, aunque para este proyecto el *boost converter* actúa como conversor AC/DC. En esencia, este conversor eleva la tensión de salida con respecto a la tensión de entrada, aunque, dependiendo del regulador, puede funcionar de una manera u otra.

El punto clave en el *boost converter* es que regulando el ciclo de trabajo, o *duty cycle* en inglés, de la señal de control, la cual se modula con un *Pulse Width Modulation*, *PWM*, se puede regular el voltaje de salida.

El convertidor elevador además suele utilizarse en sistemas de lazo cerrado, como se puede observar en la figura 2, donde se utiliza un regulador que, como se explicaba en el capítulo de introducción, hoy en día es principalmente digital debido a las ventajas que ofrece frente al regulador analógico.

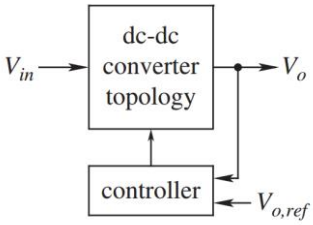

Fig. 2: Esquema en lazo cerrado de un convertidor elevador [1].

Un conversor elevador, es un ejemplo de un *corrector del factor de potencia activo* sobre el que se hablaba anteriormente, al igual que otros tipos de conversores como son el conversor reductor, o el conversor elevador-reductor.

El esquema general que siguen los conversores elevadores, así como las gráficas del comportamiento para los valores de tensión y corriente (obtenidas en su integridad de la bibliografía [1]) se muestran a lo largo de las siguientes figuras.

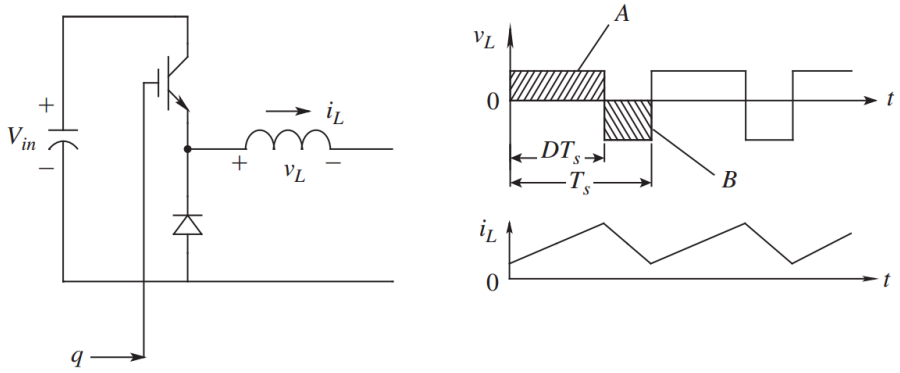

Fig.3: Diagrama básico del principio switching power pole *[1].*

Éste se basa en el principio del *switching power pole*, en donde la corriente de salida es regulada en función a la entrada, q , cuya onda es periódica, y su *duty ratio* permanece constante, D. La corriente i_L del esquema anterior está relacionada con el voltaje que pasa por el inductor, y viene determinada por:

$$i_L(t) = i_L(0) + \frac{1}{L}\int_0^t v_L \cdot d\tau$$

Dicha corriente de salida también es periódica con periodo T_s, como se puede comprobar en la figura 3 , un voltaje positivo A en voltios, incrementa la corriente i_L, así como un voltaje B negativo, hace decrecer la i_L.

Siguiendo el esquema anterior, se pueden encontrar diversos conversores, siendo los más usados el conversor elevador, conversor reductor y conversor elevador-reductor. El elegir utilizar uno u otro depende de las necesidades del sistema.

Una vez hecha esta pequeña introducción, se puede pasar a explicar a grandes rasgos el esquema de un *boost converter*.

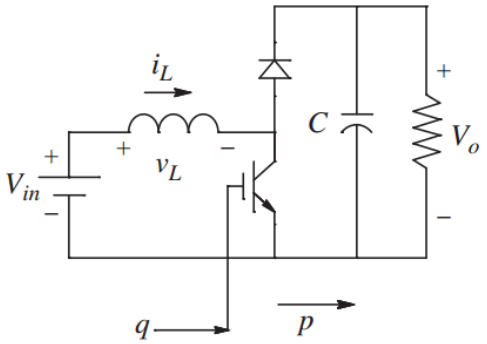

Fig.4: Esquema básico de un conversor elevador boost *[1].*

Siendo V_o el voltaje de salida, V_{in} el voltaje de entrada, q la onda periódica de *duty cycle* constante que se detalla anteriormente, y p el sentido en el que fluye la corriente.

En el *boost converter*, si el transistor está en circuito cerrado *(on)*, figura 6, hay una diferencia de tensión entre los bornes de la bobina, V_L, incrementando a su vez la corriente que circula por ella i_L, aumentando la energía en dicha bobina. Cuando el transistor pasa a estar en circuito abierto (*off*), figura 7, parte de esta energía inductiva almacenada en la bobina pasa a circular por el diodo, y por tanto hay corriente que llega a la parte derecha del circuito, formada por el condensador y la resistencia.

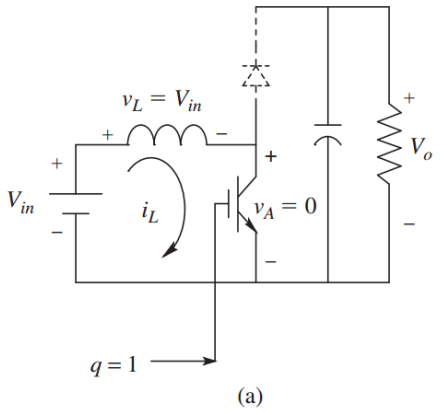

Fig.6: Funcionamiento del conversor boost para q=1 (ON). [1]

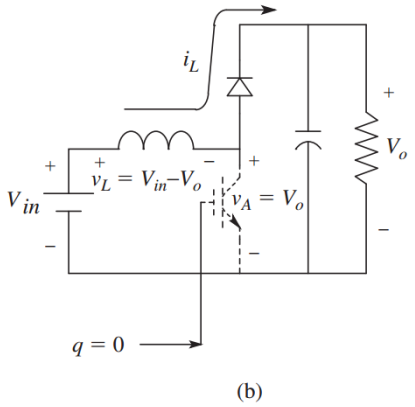

Fig.7: Funcionamiento del conversor boost para q=0 (OFF). [1]

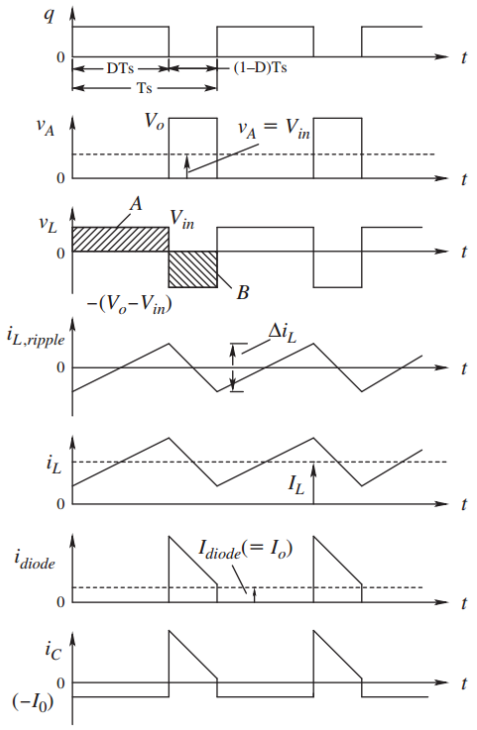

Fig.5: Voltajes y corrientes en el conversor boost. [1]

La onda periódica q de ciclo de trabajo constante D, así como el voltaje inducido en la bobina, V_L, y las corrientes que circulan tanto por la bobina como por el diodo y el condensador, pueden observarse en la figura 5 anterior.

El ratio entre voltaje de entrada y voltaje de salida, puede obtenerse o bien de la forma de onda de V_A o V_L en la figura 5 anterior, o bien usando la forma de onda del inductor, V_L, cuyo promedio es 0 en DC, se puede establecer:

$$V_{in}(DT_s) = (V_o - V_{in})(1 - D)T_s$$

Siendo la parte izquierda de la igualdad, la zona rallada A en la gráfica, y la parte derecha la zona B. Desarrollando, se llega a:

$$V_o = \frac{V_{in}}{1 - D} \qquad (V_o > V_{in})$$

Y esta es la relación que se establece entre el voltaje de entrada del conversor, y el voltaje de salida. Como se comentaba al principio, este voltaje de salida está directamente relacionada con el ciclo de trabajo de la onda periódica q conectada al transistor. Para controlar el voltaje de salida, por tanto, sólo hace falta controlar el ciclo de trabajo D.

2.3. Corrección del factor e potencia en un convertidor elevador *Boost*

El principio de operación de un PFC sobre el que se hablaba al comienzo de esta sección, una vez explicado la funcionalidad de un conversor elevador *boost*, se puede apreciar de manera bastante clara en la siguiente figura número 8:

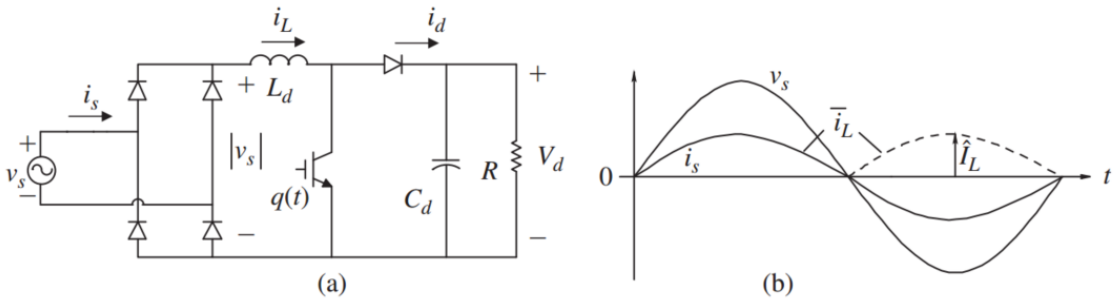

Fig. 8: Circuito PFC y su forma de onda correspondiente [1].

Se puede observar en la figura 8(a) que, entre la fuente de alimentación y el condensador C_d, se encuentra un convertidor elevador *boost*. La corriente i_L que circula a través de la bobina L_d es rectificada a onda completa, $\bar{\imath}_L(t) = \hat{I}_L|\sin \omega t|$. Como se comentaba al principio, cuanto más cercano a 1 sea el factor de potencia, menor será el desfase entre v_s y i_s, y por tanto el contenido armónico sea prácticamente nulo.

La técnica clásica de un corrector de factor de potencia se puede observar en la figura 9. En esta figura se muestran dos lazos de control, un lazo interno para regular la corriente de entrada, y otro externo para regular la tensión media de salida. Por tanto, las técnicas clásicas de corrección del factor de potencia miden 3 parámetros.

El lazo interno consigue que la corriente sea sinusoidal, tomando medida de la corriente de entrada real y comparándola con un valor de referencia. El lazo externo, por otro lado, genera un comando de potencia, para subir o bajar la potencia media [4]. El lazo de corriente funciona mucho más rápido del lazo de tensión, ya que este debe estar rectificando continuamente el valor de la corriente, para poder obtener así muchos valores que conformen una curva.

La figura 10 muestra la evolución de las tensiones de entrada y salida, corriente de entrada, y potencias de entrada y salida durante dos semiciclos de red. Como se puede observar, la potencia de entrada es variable ya que es el resultado de la multiplicación de las ondas sinusoidales v_g y i_{in} en fase. Sin embargo, la potencia de salida debe ser aproximadamente constante [4].

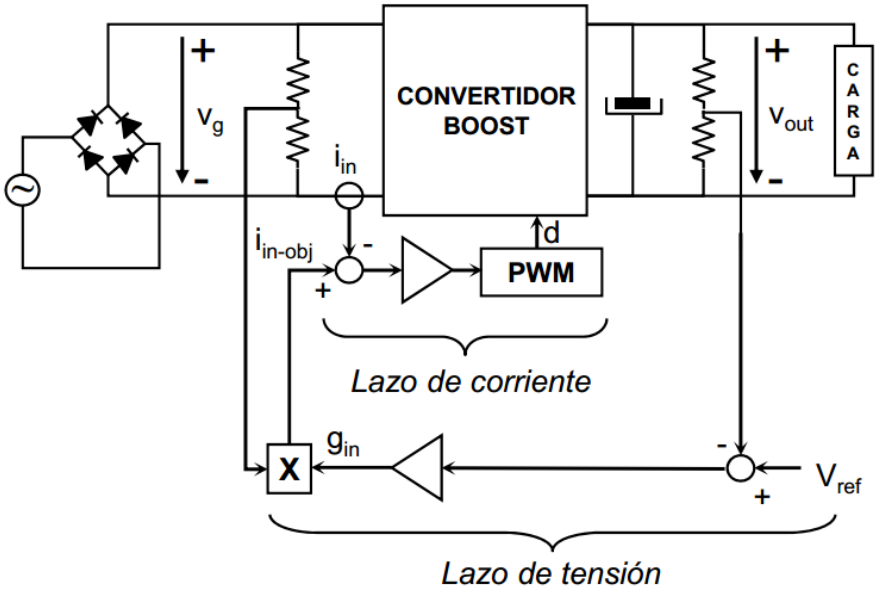

Fig. 9: Técnica PFC con un convertidor elevador elevador [3].

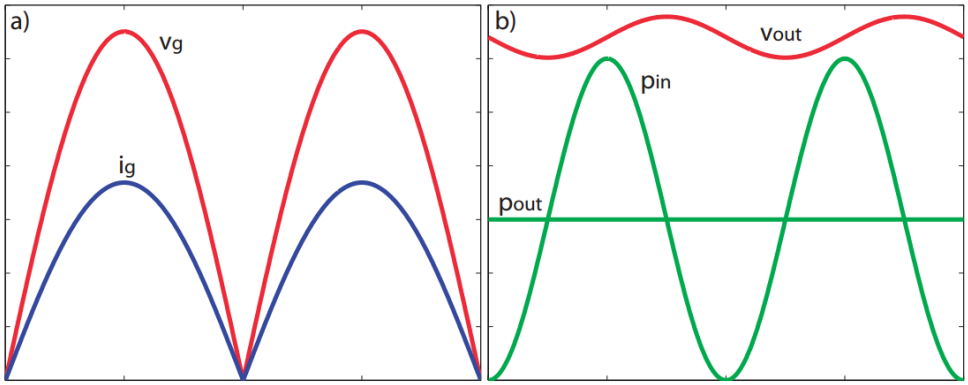

Fig. 10: a) Corriente y tensión de entrada de un convertidor PFC. b) Potencia de entrada, y tensión y potencia de salida en un convertidor PFC. [4]

3. VISIÓN COMPLETA DEL PROYECTO

En un primer acercamiento global al proyecto, se podría comenzar exponiendo que el sistema se compone de cuatro módulos principales: un extractor de datos del convertidor de potencia, una *Field Programmable Gate Array* (FGPA), un adaptador UART-USB, y una interfaz de simulación en C#.

1- Extractor de datos del convertidor de potencia. Del cual se va a obtener la información a monitorizar. Estos datos originales analógicos habrían sido convertidos a datos digitales mediante un conversor analógico digital, *ADC (analog to digital converter* en inglés*)*.
2- FPGA. Donde tendrá lugar la manipulación y transporte de la señal digital hasta la UART.
3- UART-USB. Conversor encargado de llevar la señal digital de la FPGA al puerto USB del ordenador, donde la aplicación en C# lo analizará.
4- Interfaz C#. Es la encargada de interpretar los datos que tienen origen en el convertidor elevador *Boost*.

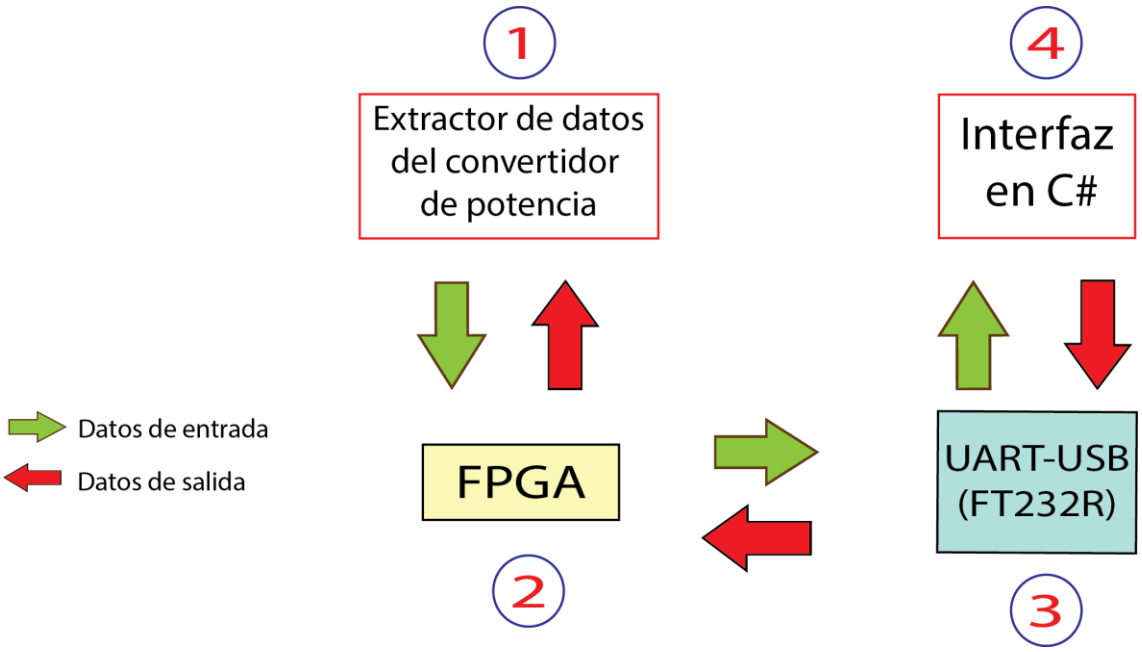

Fig. 11: Esquema general del proyecto.

A lo largo de la memoria se irá explicando más en detalle cada uno de estos módulos, pero, en general, se pueden resumir los pasos fundamentales de funcionamiento en cuatro:

- En primer lugar, el sistema se compone un convertidor de corriente alterna a corriente continua funcionando en modo de corrección de factor de potencia, el cual está llevando a cabo una conversión determinada a partir de los requerimientos del sistema. El interés reside en poder monitorizar esta conversión con la Interfaz gráfica diseñada, para observar si esta se está llevando a cabo de manera satisfactoria. Por tanto, será necesario llevar esta información analógica, convirtiéndola en información digital, hacia el puerto USB del ordenador. A su vez, este convertidor tiene incorporado un conversor analógico-digital (*ADC*), que transforma los datos analógicos de entrada en bits. Estos dos elementos son los que se denomina *extractor de datos del convertidor de potencia*, ya que los datos resultantes están listos para ser enviados y analizados a la interfaz gráfica.

- En segundo lugar se encuentra una *Field Programable Gate Array* (FPGA) que recibirá esta información del ADC y su objetivo será enviarla a la interfaz.

  En este proyecto la FPGA utilizada ha sido una FPGA de la serie Xilinx Spartan 3. En esta FPGA se ha reaprovechado el módulo UART del microprocesador *softcore PicoBlaze* [8], un pequeño procesador de 8 bits que puede ser implementado en la lógica programable de una FPGA.

  De aquí en adelante, se usará la terminología "macro/s UART" o "módulos UART" para referirse a esta sección, ya que es la misma terminología que se utiliza en la propia documentación de Xilinx.

  Es por tanto que en la FPGA tendrá lugar el envío de los datos por los puertos UART correspondientes y, para ese fin, previamente se habrá realizado la configuración de la tasa de bits apropiada, la construcción de la trama de (en este caso) 7 bytes que se va a mandar a la interfaz…También hay que remarcar que, en este sentido, la interfaz gráfica y la FPGA deben compartir un protocolo común, puesto que la interfaz debe saber en qué orden están llegando los datos. La FPGA construirá tramas de 7 bytes, donde cada trama tendrá un valor de corriente de entrada, así como de tensiones de entrada y salida, de 2 bytes cada uno. Finalmente se añadirá un *checksum* sencillo para comprobar que la trama no contiene fallos en el receptor.

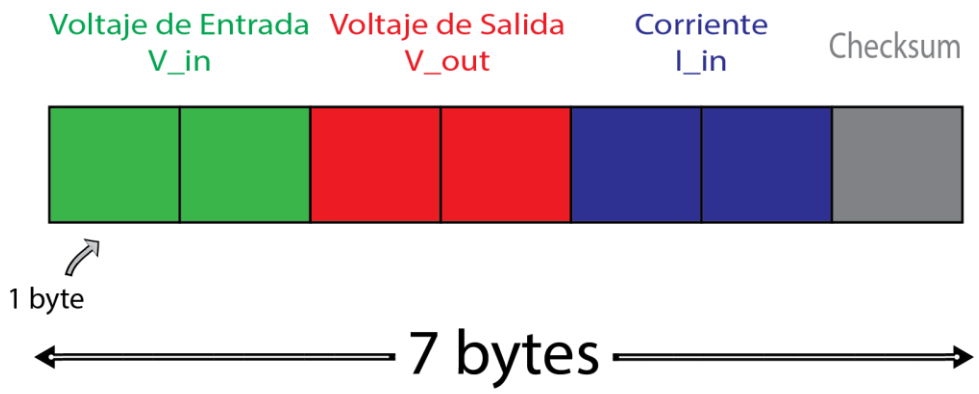

Fig. 12: Esquema de la trama de 7 bits que se manda a través de la UART.

El cálculo del *checksum* consiste en la suma de cada uno de los bytes de datos por separado.

$$Checksum = v_{in}(1) + v_{in}(2) + v_{out}(1) + v_{out}(2) + i_{in}(1) + i_{in}(2)$$

- En tercer lugar se ubica el conversor UART-USB, el cual tendrá como misión llevar la trama de 7 bytes desde el módulo UART de la FPGA, hasta el puerto USB del dispositivo. Como se explicará posteriormente, existen varios componentes que pueden desempeñar esa labor, aunque para esta implementación se ha elegido el FT232R, del fabricante FTDI.

  También se ha tenido que llevar a cabo el diseño y la construcción de una placa en la que integrar dicho dispositivo FT232R, para ello, primero se ha diseñado en Altium Designer el esquemático, posteriormente se ha impreso el circuito impreso (PCB) y finalmente se han soldado a mano los componentes.

- En cuarto y último lugar se sitúa la interfaz gráfica que en este TFG ha sido programada en lenguaje C#, utilizando *software* y librerías de Microsoft. Esta interfaz es el destino final de los distintos datos de voltajes y corrientes que se capturaron en el conversor *boost* original.

  Esta interfaz mencionada llevará a cabo un análisis y representación mediante gráficas de los datos contenidos en la trama de 7 bytes. Incorporará además distintas funcionalidades, como la capacidad de imitar el funcionamiento de un *trigger* de un osciloscopio, la selección del número de semiciclos en red a visualizar, así como el almacenaje de los datos en memoria para un posible análisis por parte de terceras aplicaciones. A su vez, esta interfaz puede ejercer de regulador, y enviar información de vuelta a través de la UART, la cual seguirá el camino

inverso de los datos que obtuvimos en primer lugar. Esta funcionalidad podría implementarse, pero no ha sido llevada a cabo en este proyecto.

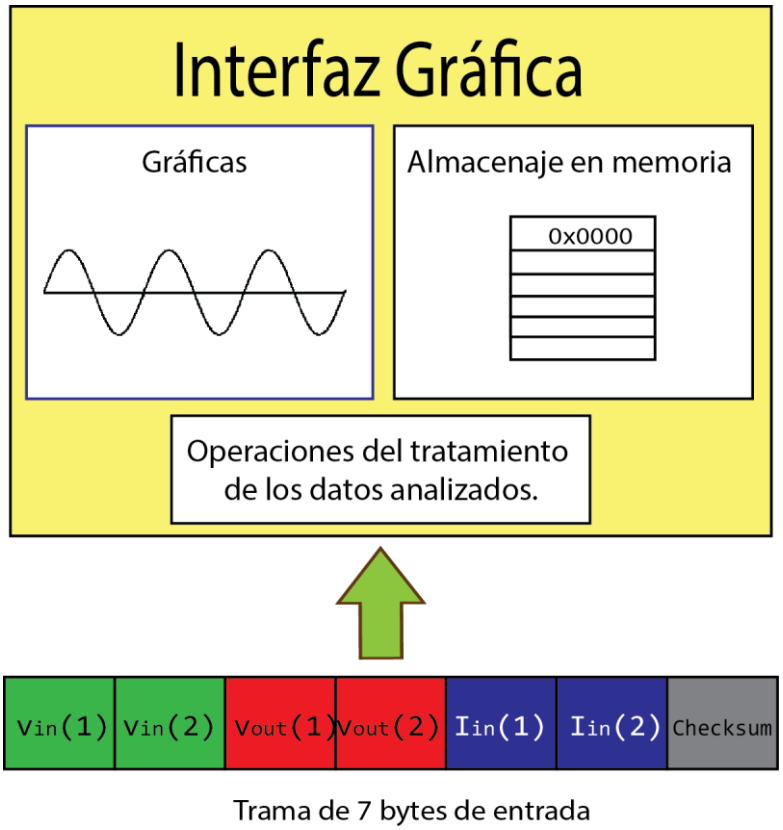

Fig. 13: Esquema de la interfaz gráfica de la aplicación.

4. MÓDULO DE COMUNICACIÓN

El módulo de comunicación engloba todos aquellos dispositivos que hacen posible la comunicación entre el convertidor elevador original del que parten los datos, y la interfaz gráfica final, que en este caso se encuentra en un PC.

4.1. Conversor Analógico Digital, ADC

Como se ha explicado anteriormente, el sistema a implementar necesita de un conversor analógico digital para convertir los valores analógicos de corriente y voltaje del *boost converter* en valores digitales que poder mandar hacia la FPGA, y posteriormente al ordenador vía módulo UART.

El regulador para el PFC que se ha explicado en el capítulo 2 anterior es digital, por lo que necesita que las magnitudes físicas (tensión de entrada, tensión de salida y corriente de entrada) a regular estén en formato digital. Para ello se utilizan ADCs. En aplicaciones de potencia se suelen utilizar ADCs de entre 8 y 12 bits [4], y en particular el sistema de PFC que se va a utilizar contiene ADCs de 10 bits.

El sistema de comunicación propuesto en este TFG prevé la utilización de ADCs de hasta 16 bits (2 bytes). En el caso de que el ADC usado contenga menos bits de resolución, el dato mandado será completado con ceros por la izquierda para no alterar su valor, mientras que si el ADC tuviese más de 16 bits de resolución, habría que eliminar los bits menos significativos sobrantes.

A continuación se va a pasar a explicar de manera muy breve los principios de conversión analógico → digital.

En resumidas cuentas, un *ADC* lleva a cabo la cuantificación y el muestreo a una frecuencia de muestreo determinada, f_s, de la señal analógica original, transformándola en un valor digital el cual representa la amplitud de esta señal original, este proceso se puede ver gráficamente en la figura 14.

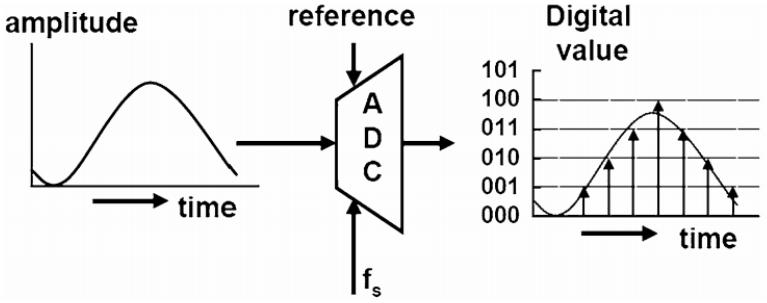

Fig. 14: Esquema básico [9] de conversión de un ADC.

Mediante la cuantificación, se asigna un margen de valor de una señal analizada a un único nivel de salida. Por otro lado, la codificación traduce estos valores obtenidos en la cuantificación a código binario.

Dependiendo de si se desea obtener más precisión o no, la cuantificación se hará con más niveles (bits) o no. En este TFG, se tiene un resultado de 10 bits, por lo que el *ADC* tendría como mínimo $2^{10} = 1024$ niveles.

Como no se va a llevar a cabo la medición de valores de tensión ni de corriente negativos, los resultados de los ADCs habría que interpretarlos como un número binario natural, por lo que el ADC generará un valor entre 0 y 1023 y, posteriormente, habrá que analizar la ganancia del ADC y el previo acondicionamiento de la señal para transformar el valor del ADC en una magnitud física.

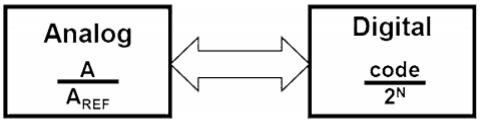

Fig. 15: Esquema básico [9] de conversión analógico/digital – digital/analógico.

4.2. FPGA: Spartan 3 *Starter Board*

Como se explicaba anteriormente, en este TFG se ha utilizado una FPGA del fabricante *Digilent*, denominada '*Spartan 3*', en su modelo *Starter Board*.

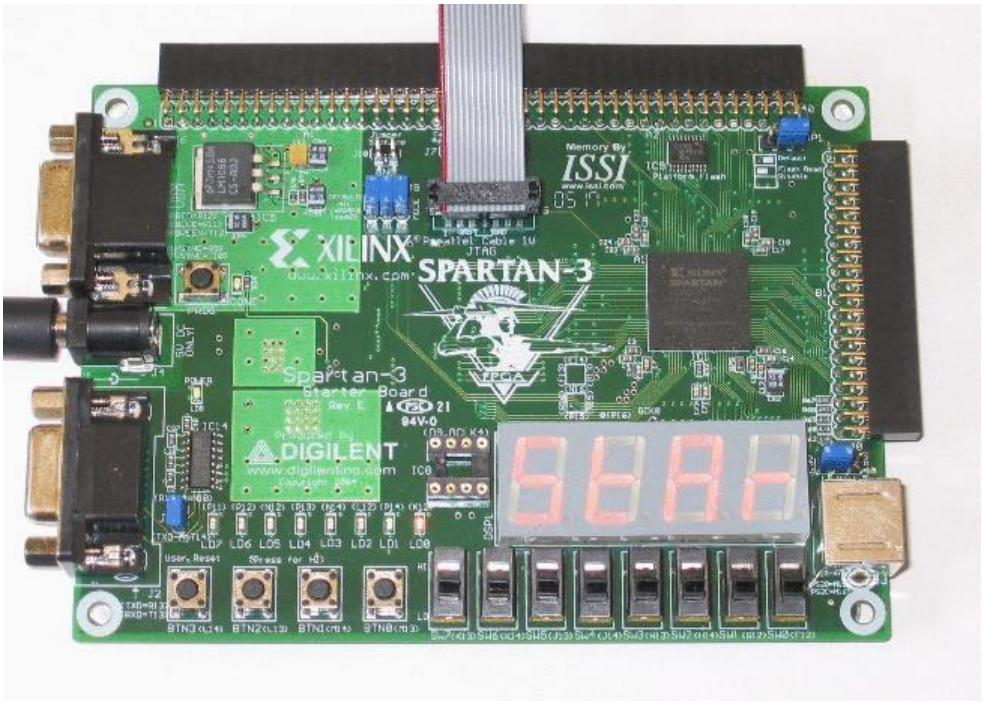

Fig. 16: FPGA Spartan 3 Starter Board utilizada en el proyecto.

Esta FPGA posee numerosas funcionalidades y características, de las cuales se explicarán las fundamentales utilizadas en el proyecto:

- La Spartan 3 contiene un gran número de *Configurable Logic Blocks* (CLB), con *Look Up Tables* (LUT) basadas en RAM, que pueden ser utilizados como *flip-flops* o *latches* y que permiten almacenar información e implementar numerosas operaciones lógicas. Estos CLB también pueden ser programados para desempeñar un rango muy amplio de operaciones lógicas.
- Bloques de *Input/Output*, IOB, que controlan el flujo de datos entre los pines de entrada y salida, y la lógica interna del dispositivo.
- RAM de 18 kbit para el almacenaje de datos.
- Digital clock Manager, DCM, para multiplicar o dividir la señal de reloj. Este bloque se analizará más en detalle tras la UART.

La FPGA posee el siguiente cometido en el proyecto: debe recibir los bits del conversor analógico digital (10 bits por dato), identificarlos, y construir la trama de 7 bytes representada en la figura 12 anterior, para mandarla luego a la interfaz en C#.

Como el estar monitorizando un convertidor *boost* constantemente de cara a hacer las pruebas para el proyecto no resulta muy práctico, ya que además el convertidor a monitorizar maneja cientos de vatios de potencia, y un fallo en el diseño de la FPGA

puede derivar en daños materiales o personales, es por ello que los datos de voltaje de entrada Vin, voltaje de salida Vout, y corriente de entrada en el *boost converter,* Iin, se han simulado y generado mediante unas memorias en código VHDL. Aún así, el probar el proyecto con el convertidor real sería algo trivial, ya que sólo habría que deshabilitar las memorias precalculadas y leer los datos del *ADC*.

Dichas memorias precalculadas funcionan de la siguiente manera: cada memoria tiene guardados los valores de Vin, Vout e Iin en 1000 puntos diferentes de un semiciclo de red. En los PFC explicados anteriormente, las señales son periódicas [4] y por tanto, se puede crear una memoria que se repita periódicamente.

A su vez, estas memorias tienen los datos en coma fija. Las señales en coma fija pueden seguir la notación QX.Y. Una señal QX.Y tiene X bits de parte entera, Y bits de parte decimal y 1 bit adicional para el signo, siguiendo la notación de complemento a dos como se puede ver en la figura 17 [4]. Por ejemplo, una señal de tipo Q6.4 tendría 6+4+1=11 bits.

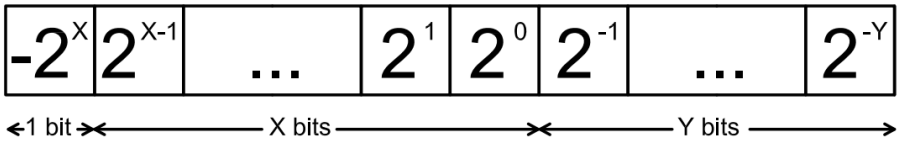

Fig. 17: Formato de una señal QX.Y [4].

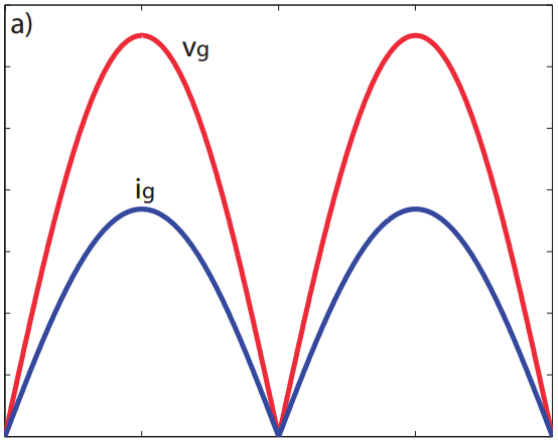

Fig. 18: Gráfica que muestra la periodicidad de las señales en un PFC explicada en la sección 3.1 [4].

```
INSTANCIACION_seno_Vin: seno_mem
port map(
    DO      => tensionEntrada,
    ADDR    => addrMem_Vin,
    CLK => Clk,
    EN      => '1',
    SSR => Reset,
    WE      => '0'
    );

INSTANCIACION_seno_Vout: seno_mem_Vout
port map(
    DO      => tensionSalida,
    ADDR    => addrMem_Vin,
    CLK => Clk,
    EN      => '1',
    SSR => Reset,
    WE      => '0'
    );

INSTANCIACION_seno_Iin: seno_mem_Iin
port map(
    DO      => CorrienteEntrada,
    ADDR    => addrMem_Vin,
    CLK => Clk,
    EN      => '1',
    SSR => Reset,
    WE      => '0'
    );
```

Fig. 19: Instanciaciones de las memorias con la que se generan los senos virtuales del boost converter.

La señal *DO* se corresponde con el valor del semiciclo generado, y *ADDR* con la dirección de memoria en la generación de la onda sinusoidal. Si se quiere tener un valor del semiciclo cada 10 µs, y puesto que se dispone de una frecuencia de reloj en la FPGA de 100 Mhz, se tiene que generar 1 dato cada 1000 ciclos, o lo que es lo mismo, hay que tener un contador que vaya hasta 1000 y una vez ahí, mande un '1' a cada *ADDR* de Vin, Vout e Iin. Este bucle se puede observar en la figura 20.

```
process (Clk, Reset)
begin
    if Reset = '1' then
        cntMemoriaAddr <= 0;
        addrMem_Vin <= (others => '0');
        addrMem_Vout <= (others => '0');
        addrMem_Iin <= (others => '0');
    elsif rising_edge (Clk) then
        -- Si queremos tener una direccion addrMem cada 10us, y nuestro Clk original es de 100Mhz,
        -- tenemos que hacer un contador hasta 1000.
        if cntMemoriaAddr = ITERACIONES_MAX_MEMORIA then
            if addrMem_Vin = conv_std_logic_vector(1000,10) then
                addrMem_Vin <= (others => '0');
            else
                addrMem_Vin <= addrMem_Vin + 1;
            end if;
            cntMemoriaAddr <= 0;
        else
            cntMemoriaAddr <= cntMemoriaAddr + 1;
        end if;


    end if;
end process;
```

Fig. 20: Código VHDL para la generación de valores de la ondas sinusoidales de Vin, Vout e Iin.

Como se puede observar en la figura 19 anterior sobre las instanciaciones de los bloques de memoria, los datos para Vin estarán contenidos en *tensionEntrada*, los de Vout en *tensionSalida* y los de Iin en *CorrienteEntrada*, para construir cada trama de 7 bytes sólo tendremos que insertar en el orden correcto estas señales en el array que deseemos utilizar, que en nuestro caso denominamos *uart_tx_frame*. Dicha construcción se muestra en la siguiente figura.

```
uart_tx_frame(0)  <= tensionEntrada(15 downto 8);
uart_tx_frame(1)  <= tensionEntrada(7 downto 0);
uart_tx_frame(2)  <= tensionSalida(15 downto 8);
uart_tx_frame(3)  <= tensionSalida(7 downto 0);
uart_tx_frame(4)  <= CorrienteEntrada(15 downto 8);
uart_tx_frame(5)  <= CorrienteEntrada(7 downto 0);
uart_tx_frame(6)  <= tensionEntrada(15 downto 8) + tensionEntrada(7 downto 0) +
                     tensionSalida(15 downto 8) + tensionSalida(7 downto 0) +
                     CorrienteEntrada(15 downto 8) + CorrienteEntrada(7 downto 0);
```

Fig. 21. Construcción de la trama a mandar por la UART a partir de los valores de tensión de entrada y salida, y de corriente de entrada.

4.3. UART

La UART, *Universal Asynchronous Receiver/Transmitter* en inglés, es uno de los puertos de comunicación de datos más extendidos. La UART permite recibir y transmitir datos en serie entre distintos dispositivos. En esencia, se necesitan por una parte una señal de transmisión de datos (TXD), una señal para la recepción de datos (RXD), y un hilo con la referencia a masa (GND). Adicionalmente, la mayoría de las UART emplean hilos complementarios para ofrecer un funcionamiento más variado.

Una comunicación asíncrona significa que el transmisor y el receptor no están sincronizados. A pesar de ello, ambos utilizan una referencia de tiempo común que hace posible la transmisión en serie de cada byte de datos.

Las siglas USART hacen referencias a dispositivos de transmisión/recepción que pueden ser tanto síncronos (con señal de reloj), como asíncronos.

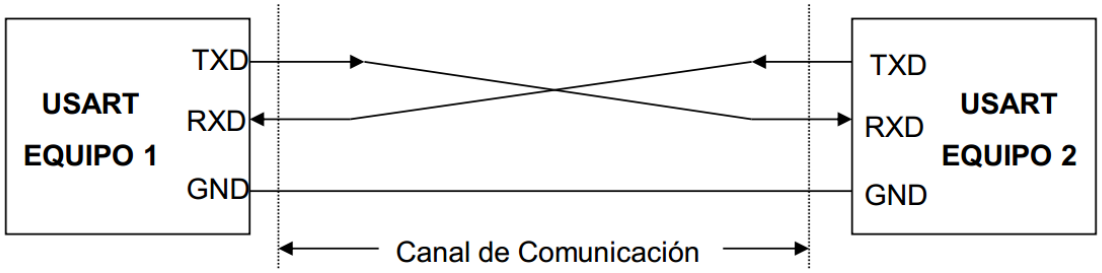

Fig.22: Comunicación serie asíncrona [10] entre 2 equipos, usando dispositivos USART, conectados en modo NULL MODEM.

La norma RS-232C es la normativa en las que se estandariza las UART. Esta norma, que es sólo válida para el modo asíncrono, define todos los elementos y características de funcionamiento del dispositivo. Dicha norma establece además, que los '0' lógicos tienen un valor de tensión entre +3 V y +25 V, así mismo, los '1' lógicos tienen un valor de -3 V a -25 V. Con dichos voltajes se consigue una importante inmunidad ante el ruido eléctrico que viaje por la masa (GND) en los equipos electrónicos, en otras palabras, se consigue disminuir el *ringing*, el *cross talk*, y otras perturbaciones debidas a la inductancia parásita de las líneas. No obstante, a pesar de ser lo más habitual, en este proyecto no se implementa esta norma, ya que la FPGA utiliza tensiones de 3,3 V, y no tiene sentido pasar a RS232, para después convertirlo a USB.

En esta TFG se van a utilizar unos módulos VHDL que implementan en lógica reprogramable una UART y que, como se comentaba anteriormente, están sacados del procesador *soft-core* PicoBlaze, que provee Xilinx. Dichos módulos son además completamente compatibles con el estándar UART previamente explicado.

Los parámetros de una transmisión/recepción sería asíncrona mediante UART deben programarse exactamente igual en ambos equipos. Dichos parámetros suelen ser, característicamente:

- Velocidad de transferencia (Baud Rate): se mide en baudios (bits por segundo). Las velocidades para el modo asíncrono están normalizadas (75, 300, 1200, 2400, 4800, 9600, 14400, 19200, 57600, 115200…). En este caso, se va a utilizar una velocidad de transferencia de 3 MBaudios, ya que como se quiere representar un sistema en tiempo real, se tiene que intentar enviar el mayor número de datos que permita el FTDI, para así conseguir una representación más precisa en la interfaz gráfica.

- Paridad: es un mecanismo sencillo de detección de errores en la comunicación. Se puede configurar una paridad par, impar, fija, o no usar paridad. En el caso del módulo UART de este proyecto, no se va a emplear paridad.

- Longitud del dato (en bits): se pueden programar varias longitudes (5, 6 7, 8 o incluso 9 bits [11]) en el proyecto desarrollado, aunque la longitud elegida finalmente va a ser de 8 bits. En esta longitud no se incluyen el bit de arranque, el bit de paridad ni los bits de parada.

- Bit de parada (STOP): se pueden seleccionar 1, 1,5 y 2 bits de parada para el modo asíncrono. Los bits de parada son siempre '1' y sirven para indicar el fin del carácter transmitido. La UART utilizada en este caso, usará tanto 1 bit de STOP, como 1 bit de START.

En resumen:

Tabla 1: Datos de los parámetros de transmisión/recepción del módulo UART.

	Módulo Xilinx UART
Start	1 bit
Datos	8 bits
Paridad	Sin paridad
Stop	1 bit
Velocidad	3.000.000 baudios

Esta UART, como se ha explicado anteriormente, transmite los datos de manera asíncrona. Estos datos son transmitidos mandando el LSB primero, a la velocidad de transmisión que se configure. El receptor necesita poder detectar que alguien le está mandado el primer bit LSB de información, para ello, el transmisor envía un señal de s*tart* activo bajo por la duración de 1 bit al receptor.

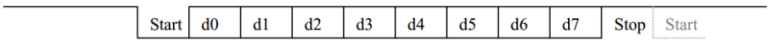

Fig.23: Transmisión en serie de bits en la UART [12].

El receptor utiliza el flanco de bajada del bit de *start* para comenzar el cronometrado interno del circuito. Este cronometrado es luego usado para muestrear el valor de los

bits de entrada, aproximadamente en la mitad de la duración del pulso del bit, ya que este es el punto en el que dicho valor es más estable. Después de que se haya muestreado el último bit de datos (MSB), el receptor comprueba si el transmisor ha mandado el bit de parada (*stop bit*), o continúa la transmisión.

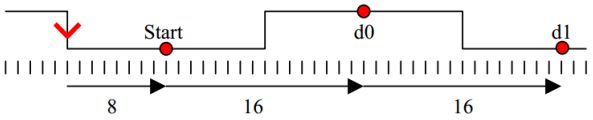

Fig.24: Muestreo de los datos de entrada en el receptor de la UART [12].

Debido a que el receptor re-sincroniza (reinicia el cronometrado interno del circuito) en el flanco de bajada de cada bit de comienzo, la sincronización entre transmisor y receptor sólo necesita ser igual a una tolerancia de medio periodo de bit cada 10 bits. Este error del 5% es tolerable y no debería suponer una complicación en ningún sistema digital. Los módulos VHDL a utilizar, esperarán que la referencia de tiempo se provea con forma de señal de *enable*, la cual tiene una velocidad 16 veces la tasa de transmisión en baudios, *baud rate*.

Esta señal 16 veces más rápida que el *baud rate* se utiliza para crear con precisión las tramas a la velocidad pedida, es denominada *en_16_x_baud* (ya que la señal se utiliza como habilitación de reloj dentro de las macros), y además debe ser provista al reloj de manera síncrona, y tener un pulso de duración 1 ciclo de reloj.

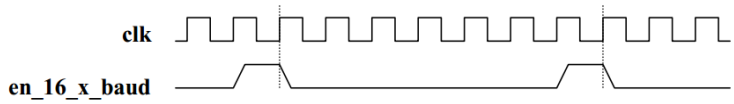

Fig. 25: Forma de onda de en_16_x_baud con respecto al reloj clk de referencia [12].

El reloj físico proveniente de la FPGA es de 50 Mhz, aunque el regulador para el PFC requiere un reloj con frecuencia 100 Mhz. De estos 100 Mhz se puede intentar obtener la frecuencia de *en_16_x_baud*. Como que se desea obtener una velocidad de transmisión de 3.000.000 bits por segundo, y esta señal debe ir 16 veces a la velocidad de transmisión:

$$\frac{100\ Mhz}{16\ x\ 3.000.000} = 2{,}08333\ \approx 2$$

Se necesitaría que por cada 2 pulsos del clk del regulador, se tuviese 1 pulso en la señal *en_16_x_baud*. O lo que es lo mismo, *en_16_x_baud* tendrá una velocidad aproximada de 50 Mhz. Con la aproximación que se está haciendo en los cálculos, aproximando 2,0833 a 2, se comete un error de un 4,2%. En pruebas experimentales con la interfaz gráfica, se ha comprobado que este error del 4,2% para una tasa de 3 Mbaudios es excesivo, ya que la mayoría de las tramas llegan corruptas o con datos inválidos, y el *checksum* es inválido. La frecuencia real utilizada en la señal *en_16_x_baud* se puede observar en la figura 26. Más adelante se explicará cómo obtener estos 48 Mhz de manera exacta en el proyecto en VHDL.

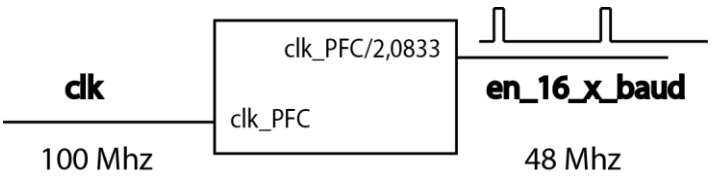

Fig. 26: Velocidad real obtenida en en_16_x_baud en función al reloj de referencia de 100 Mhz del PFC.

Debido a los problemas de precisión anteriormente expuestos, se ha decidido utilizar en el proyecto un *Digital Clock Managar*, DCM, para obtener una mayor precisión en la señal *en_16_x_baud*. En la figura 27, se muestra un fragmento del código mediante el cual se obtiene la frecuencia deseada en *en_16_x_baud*, la señal *clk_UART*, es la señal que proviene del DCM anteriormente mencionado, y que nos provee de la frecuencia exacta para que nuestro sistema pueda trabajar a 3.000.000 Baudios.

```
process(Clk, Reset)
begin
   if Reset = '1' then
      enable16x_uart <= '0';
      clk_UARTR <= '0';
   elsif rising_edge(Clk) then
      clk_UARTR <= clk_UART;
      if clk_UART = '1' and clk_UARTR = '0' then
         enable16x_uart <= '1';
      else
         enable16x_uart <= '0';
      end if;
   end if;
end process;
```

Fig. 27: Código VHDL para obtener la frecuencia deseada en 'en_16_x_baud'.

En posteriores secciones se explicará más en detalle el funcionamiento de un DCM, así como su implementación en el proyecto de Xilinx ISE.

Durante la transmisión de bits, el estado normal de la línea por donde circulan estos bits es activo alto, por lo tanto un nuevo bit de *start* es identificado mediante un flanco de bajada. Es debido a esta peculiaridad que, si el transmisor envía un flanco de bajada antes de la finalización de los 8 bits de la trama, se considera que se está produciendo una condición de *break*, la cual puede ser debida a distintos motivos, como que el transmisor se ha quedado sin suministro eléctrico por ejemplo. En este caso, el receptor considerará los datos de la trama recibidos inválidos y los descartará. Asimismo, la UART receptora esperará a que la línea vuelva a transmitir '1', y se reciba un flanco de bajada correspondiente a un bit de *stop* válido. Esta explicación se ilustra en la figura 28.

Fig. 28: Resolución de la situación de 'break condition' por parte de la UART receptora. [12].

4.3.1) Transmisión en la UART: UART_TX

En primer lugar, habría que decir que tanto los módulos VHDL de transmisión, UART_TX, y de recepción, UART_RX, contienen un buffer *FIFO* de 16 bytes embebido.

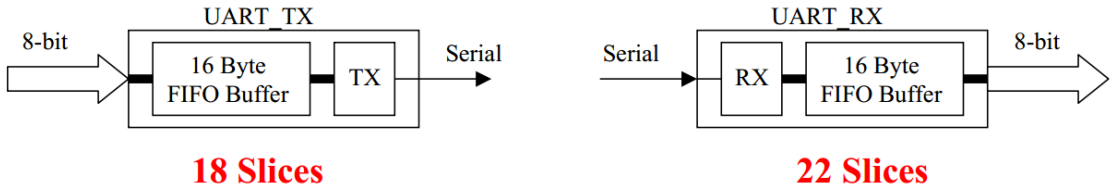

Fig. 29: Buffers embebidos de las macro UART. Tanto para transmisión (izquierda) como para recepción (derecha). [12].

UART_TX hace alusión a la parte de transmisión de la UART, así como UART_RX hace mención a la parte de recepción. Es habitual encontrar las siglas TX para hacer referencia a transmisión, y RX, para hacer referencia a recepción.

Estos buffers soportan tasas a partir de 9600 Baudios y permiten que el sistema de transmisión pueda dar la orden de envío de una trama de hasta 16 bytes sin tener en cuenta la transmisión UART, siempre que estas tramas no estén demasiados seguidas. En este caso, como las tramas utilizadas son de 7 bytes, este buffer permite esa ventaja, el dar la orden de envío de toda la trama a la vez (realmente 1 byte de ciclo por reloj), independientemente de la velocidad de la UART.

El módulo VHDL de transmisión de la UART, UART_TX, viene implementado mediante tres archivos VHDL en el proyecto Xilinx. El *top level* se denomina *uart_tx.vhd*, el cual combina los módulos del buffer *FIFO bbfifo_16x8.vhd* y el transmisor compacto UART constante (k) *kcuart_tx.vhd*. [12]

Las señales que componen dicho módulo VHDL, vienen representadas en la figura 30:

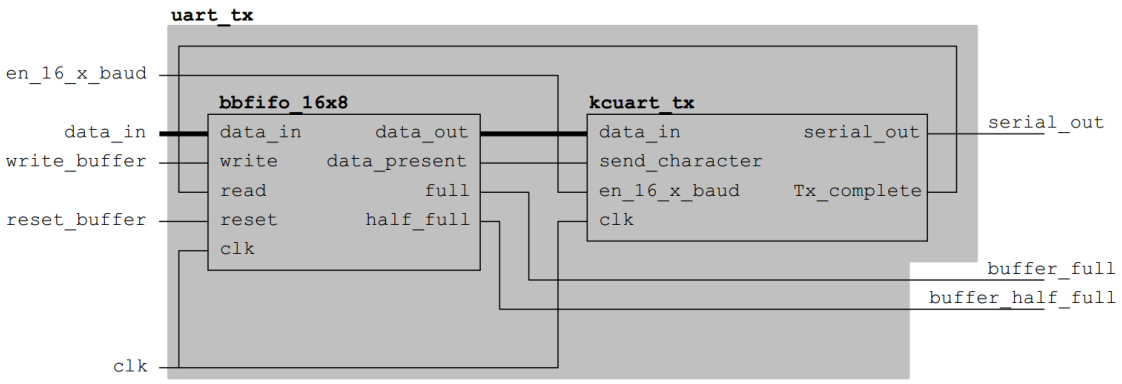

Fig. 30: Esquema de las señales del top level del módulo UART_TX. [12].

Las cuales vienen definidas en VHDL de la siguiente manera:

```
COMPONENT uart_tx is
    port (
                  data_in : in std_logic_vector(7 downto 0);
            write_buffer : in std_logic;
            reset_buffer : in std_logic;
            en_16_x_baud : in std_logic;
              serial_out : out std_logic;
             buffer_full : out std_logic;
        buffer_half_full : out std_logic;
                     clk : in std_logic

    );
end COMPONENT;
```

Fig. 31: Código VHDL de las señales del módulo UART_TX.

A continuación, se va a pasar a explicar el significado de las siguientes señales de protocolo hardware para usar la UART, haciendo hincapié en aquellas que más se utilizan en este TFG.

`data_in` – 8 bits de entrada que van a ser transmitidos en serie. El dato se captura por el buffer *FIFO* con un flanco de subida del reloj, estando ***`write_buffer`*** activado.

`write_buffer` – Como se acaba de explicar, cuando se pone a '1' el dato de entrada ***`data_in`***, este se escribe en el buffer. Tendrá lugar por tanto una operación de escritura en cada ciclo de reloj en el que esté activa esta señal. Si el buffer *FIFO* se llena porque la tasa de entrada de datos (provocada por ***`write_buffer`*** igual a '1') es mayor que la tasa de salida (determinada por la velocidad de la UART), el resto de datos que sigan llegando se ignorarán.

`reset_buffer` – Activo en alto. Sirve para inicializar el módulo transmisor.

`en_16_x_baud` – Señal de *enable* con una frecuencia 16 veces mayor que la velocidad de transmisión deseada. Esta señal ha sido explicada anteriormente.

`serial_out` – Señal de salida del protocolo UART.

`buffer_full` – Cuando el buffer se encuentra lleno, esta salida se pone a '1'. Cualquier dato que se intenta escribir en el buffer mientras esta señal esté a '1' será descartado. Esta señal no se usa en este TFG ya que la tasa de envío ha sido calculada para evitar problemas de llenado con el buffer, tal y como se observa en la sección que se habla del buffer de transmisión.

`buffer_half_full` – Esta salida se activa a '1' cuando el buffer contiene más de 8 bytes de datos. En otras palabras, sirve como aviso para que la lógica de envío externa a este módulo pueda saber que hay riesgo de llenado en el buffer. Al igual que la señal `buffer_full` esta señal no ha sido necesario utilizarla.

`clk` – Señal utilizada por todos los elementos síncronos de la macro.

```
INSTANCIACION_uart_tx: uart_tx
port map (
                    data_in => DATA_In_uart_tx,
                write_buffer => write_buffer_tx,
                reset_buffer => Reset,
                en_16_x_baud => enable16x_uart,
                  serial_out => Serial_out,
                 buffer_full => buffer_full_tx,
            buffer_half_full => buffer_half_full_tx,
                         clk => Clk

);
```

Fig. 32: Instanciación del módulo UART_TX.

Es por tanto que en `data_in` se tiene que ir mandado las tramas de 7 bytes que estamos generando en la FPGA, byte a byte. Si se observa la figura 32 anterior, se ve que se denomina a `DATA_In_uart_tx` en el código VHDL como la señal de 8 bits por los que se va a ir mandando los bytes de la trama en serie. Este envío va a ser realizado implementando una máquina de estados, *FSM*, de 1 estado por cada byte que se necesite mandar. Esta máquina de estados se encontrará por defecto en un estado de reposo, *IDLE*, esperando a que se le envíe una notificación de que la trama está lista para enviar.

Cuando se active el *flag* que indica que nuestra trama está lista para enviar, se irá recorriendo uno a uno cada estado de la FSM, y depositando cada byte de la trama en el puerto de transmisión de la UART_TX.

A continuación se muestran diversas figurar para ilustrar esta máquina de estados. En la figura 33 se observa un esquema general de dicha máquina de estados, y en las figuras 34 y 35 sus correspondientes códigos VHDL implementados.

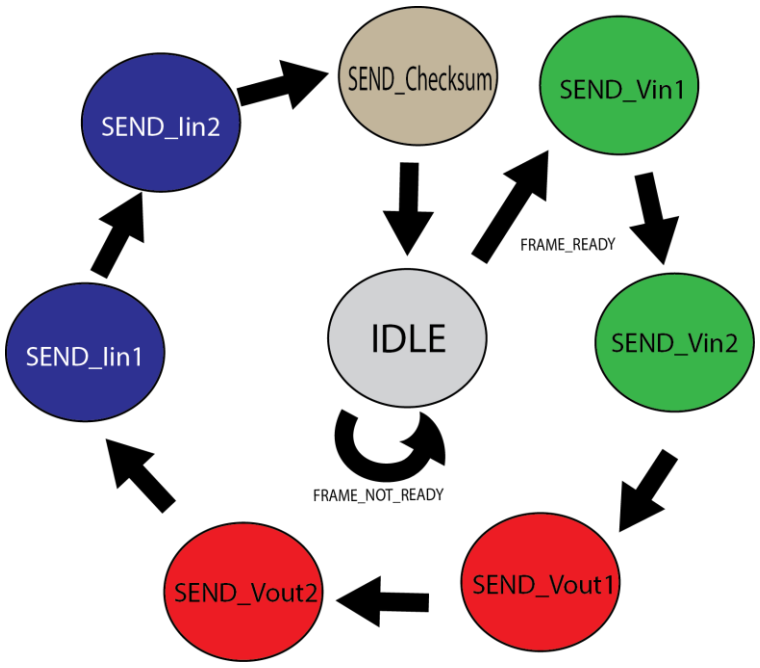

Fig. 33: Esquema general de la máquina de estados para la construcción de la trama a enviar por UART_TX.

```
process (Clk, Reset)
begin
    if Reset = '1' then
        Estado <= IDLE;
        write_buffer_tx <= '0'; --

    elsif rising_edge(Clk) then
        case Estado is
            when IDLE =>
            -- En este estado nos encontramos esperando a que el bit uart_tx_frame_ready
            -- nos indique que hemos recibido datos para enviar.
                if uart_tx_frame_ready = '1' then
                    Estado <= SEND_Vin1;
                else
                    Estado <= IDLE;
                    DATA_In_uart_tx <= (others => '0');
                    write_buffer_tx <= '0';

                end if;
```

Fig. 34: Código VHDL del estado de reposo IDLE de la máquina de estados que se encuentra a la espera de recibir una trama válida para enviar.

```
                when SEND_Vin1 =>
                        write_buffer_tx <= '1';
                        DATA_In_uart_tx <= uart_tx_frame(0);

                when SEND_Vin2 =>
                        DATA_In_uart_tx <= uart_tx_frame(1);
                        Estado <= SEND_Vout1;


                when SEND_Vout1 =>
                        DATA_In_uart_tx <= uart_tx_frame(2);
                        Estado <=SEND_Vout2;

                when SEND_Vout2 =>
                        DATA_In_uart_tx <= uart_tx_frame(3);
                        Estado <= SEND_Iin1;

                when SEND_Iin1 =>
                        DATA_In_uart_tx <= uart_tx_frame(4);
                        Estado <= SEND_Iin2;

                when SEND_Iin2 =>
                        DATA_In_uart_tx <= uart_tx_frame(5);
                        Estado <= SEND_CRC;

                when SEND_CRC =>
                        DATA_In_uart_tx <= uart_tx_frame(6);


                        Estado <= IDLE;

            end case;
        end if;
    end process;
```

Fig. 35: Código VHDL del resto de la máquina de estados, donde se lleva a cabo el envío byte a byte de la trama.

4.3.2) Buffer de transmisión

El funcionamiento del buffer cuando se escriben datos en él se puede apreciar en el la siguiente figura número 36. El buffer se lee automáticamente por el circuito *kcuart_tx* y se pasan dichos datos a la línea serie por la que se transmiten al receptor.

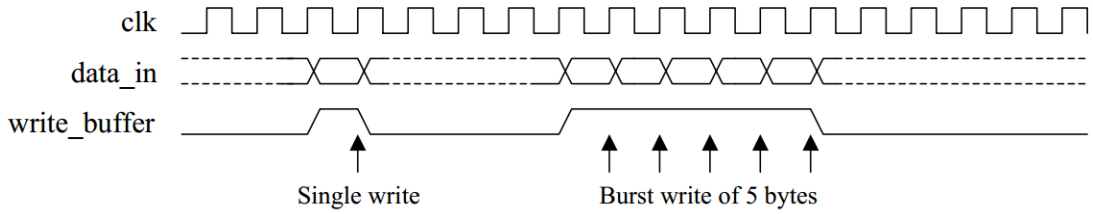

Fig. 36: Escritura de datos en el buffer de UART_TX. [12]

En el proyecto implementado, se aprovecha esta funcionalidad de poder escribir ráfagas de bytes en el buffer para escribir los 7 bytes de la trama.

Cuando el buffer se llena, este es el diagrama de ondas que se puede apreciar:

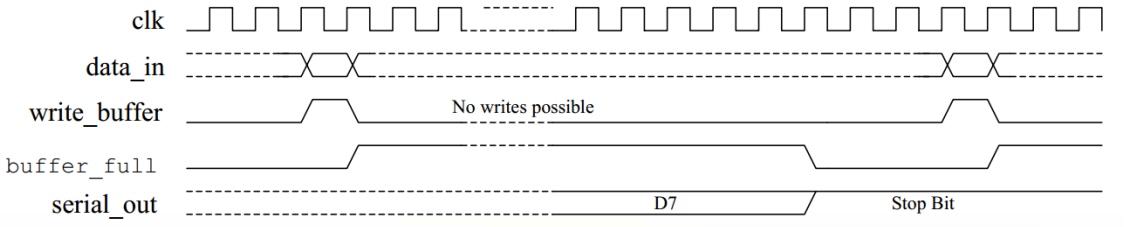

Fig. 37: Diagrama de ondas en el buffer de UART_TX cuando este se encuentra lleno. [12]

En el sistema propuesto, este caso nunca llega a darse debido a lo que se ha explicado anteriormente: la tasa de envío de las tramas es inferior a la tasa de salida, y por tanto no se satura el buffer.

Si no hubiera buffer, no se podría dar la orden de enviar un byte hasta que no se hubiese enviado el anterior. Al tener un buffer, en 7 ciclos consecutivos de reloj de la FPGA se puede dar la orden de envío de los 7 bytes de la trama. Sin embargo, hay que tener cuidado porque el buffer podría llenarse. En este proyecto, si se observa el funcionamiento de la máquina de estados, esto no llega a ocurrir nunca, ya que el buffer es de 16 bytes, y al ser la trama de 7 bytes, la máquina de estados no empieza a mandar datos de la trama nueva hasta que no se termine de enviar la trama anterior.

4.3.3) Recepción en la UART: UART_RX

Aunque, como se desarrolló anteriormente, el sistema desarrollado sólo prevé la comunicación FPGA → PC, se ha implementado también un módulo UART de recepción, UART_RX, para posibles ampliaciones en la configuración del sistema de comunicación. Es por ello que a continuación se va a explicar brevemente este módulo.

Al igual que para el caso de UART_TX, UART_RX dispone de 3 archivos VHDL: el *top level*, *uart_rx.vhd*, el cual instancia al buffer de recepción *bbfifo_16x8.vhd* y al receptor UART, *kcuart_rx.vhd*.

Las señales que componente esta macro son:

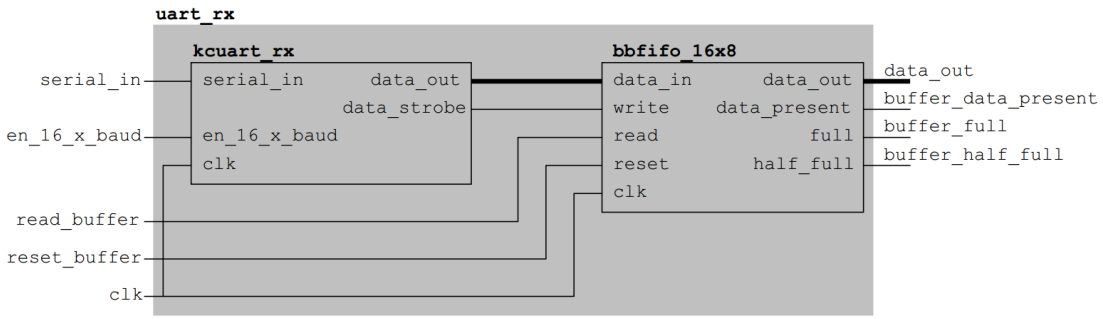

Fig. 38: Esquema de las señales del top level del módulo UART_RX. [12]

Y viene definido en el código VHDL de la siguiente manera:

```
COMPONENT uart_rx is
   port (
                  serial_in : in std_logic;
                   data_out : out std_logic_vector(7 downto 0);
                read_buffer : in std_logic;
               reset_buffer : in std_logic;
               en_16_x_baud : in std_logic;
        buffer_data_present : out std_logic;
                buffer_full : out std_logic;
           buffer_half_full : out std_logic;
                        clk : in std_logic

   );
end COMPONENT;
```

Fig. 39: Código VHDL de las señales del módulo UART_RX.

De manera análoga que para el caso UART_TX, se va a pasar a analizar el significado de estas señales:

`serial_in` – Datos de entrada provenientes de UART_TX. Estos están formados por el bit de *start* y *stop*, y los 8 bits de datos, empezando por el LSB. Como se comentó anteriormente, el flanco de bajada del bit de *start* se usa para identificar el comienzo de la transmisión de datos en serie.

data_out – Los 8 bits de datos que se han recibido. Este dato es válido cuando la señal *buffer_data_present* está activa.

read_buffer – Cuando se activa a '1' indica que los datos que contienen *data_out* han sido leídos (o van a ser leídos en el siguiente pulso de reloj) y que por tanto el buffer *FIFO* tendría que hacer disponible la siguiente trama de datos.

reset_buffer – Activo alto. Sirve para inicializar el módulo receptor.

en_16_x_baud – Señal de *enable* con una frecuencia 16 veces mayor que la velocidad de transmisión deseada.

buffer_data_present – Cuando el buffer interno contiene uno o más bytes de datos recibidos, este *flag* se pondrá a '1', y los datos válidos estarán disponibles en el puerto *data_out*.

buffer_full – Activo alto. Indica cuando los 16 bytes del buffer se encuentran llenos. Como se ha comentado anteriormente, esta señal no se utiliza.

buffer_half_full – Activo alto. Indica si hay 8 o más bytes del buffer en uso. Esta señal tampoco se utiliza en este TFG.

4.3.4) Buffer de recepción

El buffer de recepción funciona de manera bastante análoga al buffer de transmisión, con los pequeños cambios que se han explicado en las señales que lo componen.

Las figuras 40 y 41 muestran los distintos diagrama de ondas, en primer lugar para el caso en que se esté leyendo los datos recibidos por la transmisión serie, y en segundo lugar para el caso en que el buffer de 16 bytes se llene.

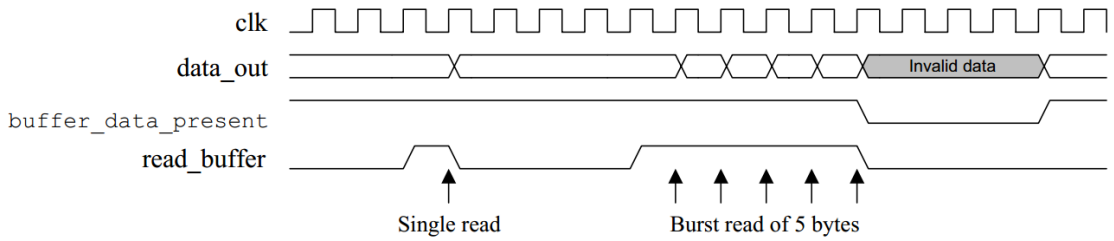

Fig. 40: Lectura de datos en el buffer de UART_RX. [12]

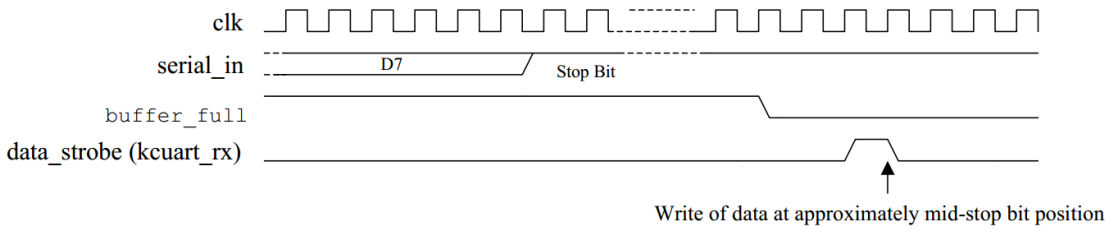

Fig. 41: Diagrama de ondas en el buffer de UART_RX cuando este se encuentra lleno. [12]

4.4. *Digital Clock Manager*, DCM

En esencia, un *Digital Clock Manager,* multiplica o divide la señal original de reloj para sintetizar una frecuencia nueva. Los DCM también ayudan a reducir los problemas de *skew* del reloj.

La Spartan 3 contiene un total de cuatro DCM, y estos se integran directamente en la red de distribución del reloj en la FPGA siguiendo el esquema de la siguiente figura:

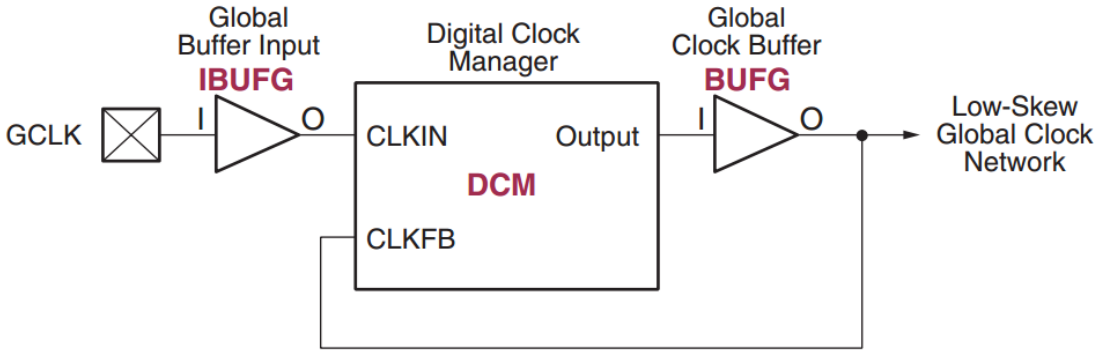

Fig. 42: El DCM se inserta directamente en la red de distribución del reloj. [9]

Como se explicó con anterioridad, en este proyecto se ha necesitado sintetizar un DCM para conseguir una frecuencia más precisa para la señal `en_16_x_baud` así como para el reloj del regulador para el PFC, y así reducir las pérdidas.

La FPGA Spartan 3 tiene un reloj físico de 50 Mhz, el cual es multiplicado por un DCM por 2 para obtener la frecuencia de 100 Mhz para el regulador PFC.

Por otro lado, es necesario disponer de otro DCM para poder sintetizar la frecuencia teórica de 48 Mhz de la señal que controla el cronometrado de los módulos UART `en_16_x_baud`. Recordando los cálculos, esta señal tenía que tener 1 pulso por cada 2,08333 pulsos aproximadamente de la señal del reloj físico original de la FPGA si se quiere ir a 16 veces la tasa de transmisión (*baud rate*).

$$\frac{100Mhz}{16\ x\ 3.000.000} \approx 2{,}083333333$$

Este resultado significa que, `en_16_x_baud` tendría una frecuencia de:

$$\frac{100\ Mhz}{2{,}083333333} = 48{,}00000001\ Mhz$$

Y por tanto, sólo se podría sintetizar esta frecuencia con uno de los *digital clock manager* integrados en la Spartan-3.

La interfaz para la configuración de dicho DCM en Xilinx se puede encontrar en la figura 43, y su correspondiente código VHDL en la figura 44.

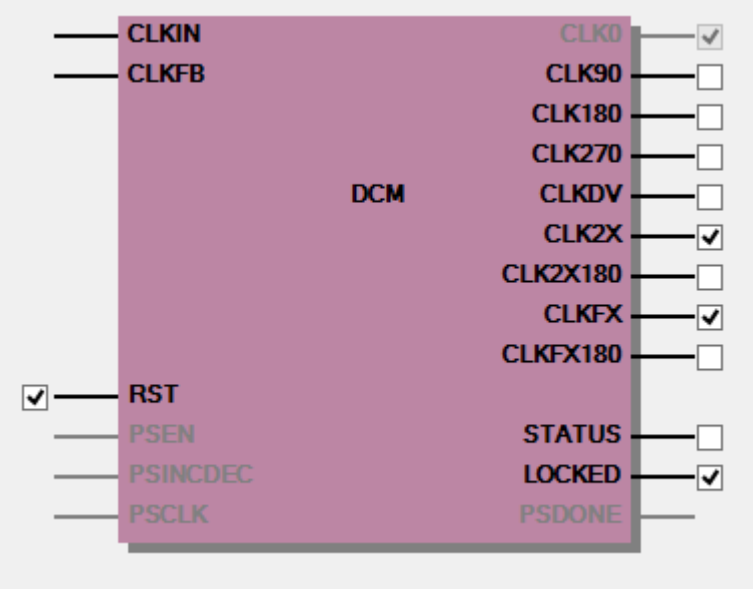

Fig. 43: Interfaz de configuración del DCM en Xilinx.

```
COMPONENT mult_reloj
PORT(
    CLKIN_IN : IN std_logic;
    RST_IN : IN std_logic;
    CLKFX_OUT : OUT std_logic;
    CLKIN_IBUFG_OUT : OUT std_logic;
    CLK0_OUT : OUT std_logic;
    CLK2X_OUT : OUT std_logic;
    LOCKED_OUT : OUT std_logic
    );
END COMPONENT;
```

Fig. 44: Código VHDL de las señales del DCM en Xilinx.

Siendo la señal `CLKFX` la salida con el valor deseado para la nueva frecuencia sintetizada, y que contiene estos valores:

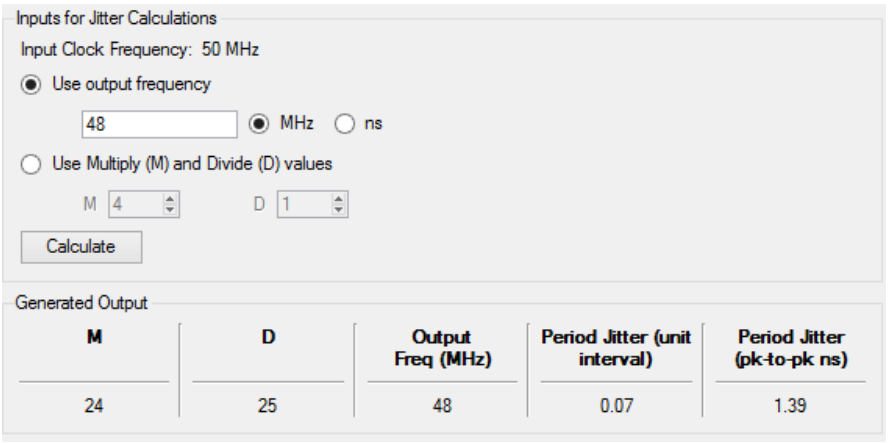

Fig. 45: Cálculo de los valores para el DCM en Xilinx.

Puesto que es más fácil obtener un reloj de 48 Mhz a partir de uno de 50 Mhz que a partir de uno de los 100 Mhz usados para el convertidor, se han usado estos 50 Mhz por el DCM, el cual los ha multiplica por 24 (M) y lo divide entre 25 (D), obteniendo así exactamente los 48 Mhz deseados.

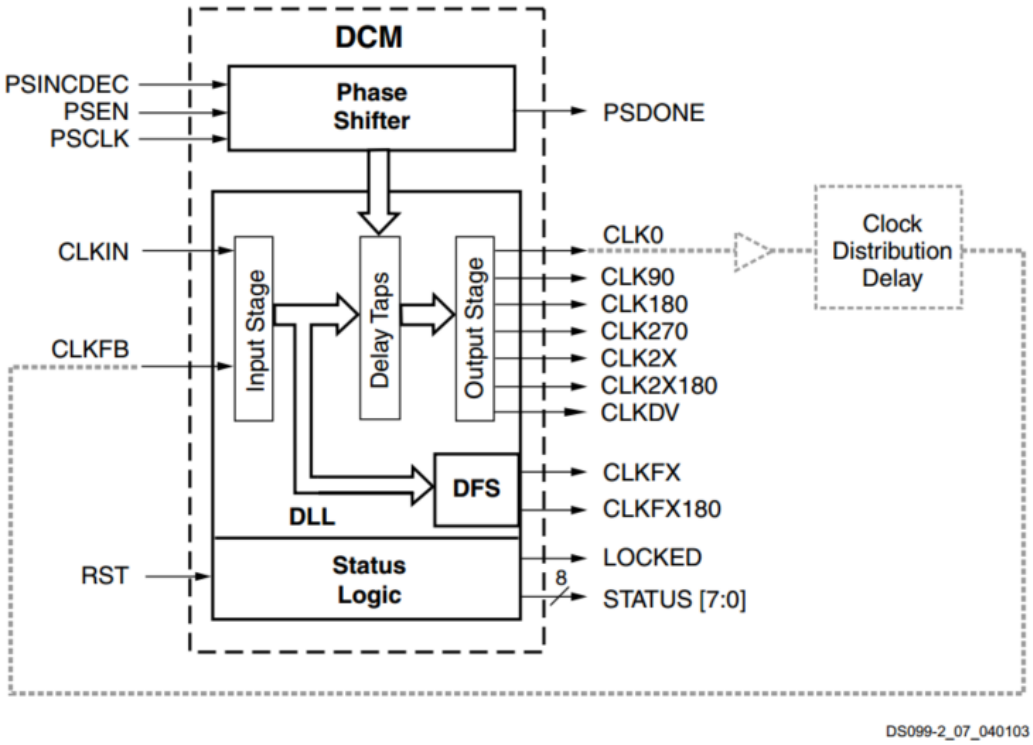

Fig. 46: Bloque funcional del DCM en la Spartan 3. [13]

5. MÓDULO UART-USB

El módulo UART-USB engloba todos aquellos procedimientos seguidos de cara a construir el circuito impreso, PCB, necesario para completar el módulo de comunicación desarrollado en el capítulo anterior.

En este módulo además se hace un detalle técnico más pormenorizado del dispositivo elegido para llevar a cabo dicha tarea de comunicación con la interfaz gráfica.

5.1. Introducción: construcción del FT232R en Altium Designer

Como se observó en el esquema inicial, es necesario disponer de un circuito de conversión de señales para conectar la FPGA con un ordenador.

Por una parte, si se utilizase un puerto serie tradicional de ordenador, el estándar usado sería el RS232, que conlleva unos niveles de tensión (+3 a 15 voltios y 3 a -15 voltios), que no soporta la FPGA. Una opción sería utilizar un chip de adaptación a esos niveles, como es el MAX232. Sin embargo, se ha optado por realizar una conversión UART-USB, utilizando para ello el dispositivo FT232R que se desarrollará a continuación.

5.2. Comparación de distintos dispositivos USB-to-UART

En primer lugar se va a realizar una comparación de dispositivos USB-UART de distintos fabricantes que se pueden encontrar en una tienda online de componentes electrónicos como puede ser Farnell [14], para justificar la elección del dispositivo FT232R en este TFG frente a otros dispositivos disponibles en el mercado.

Los componentes a comparar son el MCP2200-I/SO del fabricante MICROCHIP, el CY7C65213-32LTXI de CYPRESS SEMICONDUCTOR, el XR21B1421IL28-F de EXAR, y por último el que se ha utilizado para la implementación del proyecto, el FT232RL del fabricante FTDI.

Tabla 2: Comparativa de distintos dispositivos USB to UART en Farnell. [14]

Código Farnell	1781148	1146032RL	2345648	2454869
Imagen del producto				
Fabricante y referencia	MICROCHIP MCP2200-I/SO	FTDI FT232RL	CYPRESS SEMICONDUCTOR CY7C65213-32LTXI	EXAR XR21B1421IL28-F
RoHS	Sí	Sí	Sí	
Descripción	MICROCHIP-MCP2200-I/SO-IC, USB2.0-TO-UART, W/GPIO, 20SOIC	FTDI-FT232RL-IC, USB A UART, SMD, 28SSOP	CYPRESS SEMICONDUCTOR-CY7C65213-32LTXI-IC, USB-SERIAL BRIDGE CNTRL, 1CH, 32QFN	EXAR-XR21B1421IL28-F-PUENTES INTERFAZ, USB A UART, QFN-28
Precio	1,93 €	3,52 €	2,15 €	3,74 €
Precio para	Unidad 1	Cinta y rollo cortados 1	Unidad 1	Unidad 1
Modelo de Interfaz	SOIC	SSOP	QFN	QFN
Nivel de Sensibilidad a la Humedad (MSL)	MSL 1 - Ilimitado	MSL 3 - 168 horas	MSL 3 - 168 horas	MSL 2 - 1 Año
Núm. de Contactos	20	28	32	28
Sustancia Extremadamente Preocupante (SVHC)		No SVHC (16-Jun-2014)	No SVHC (16-Jun-2014)	To Be Advised
Temperatura de Trabajo Máx.	85	85	85	85
Temperatura de Trabajo Mín.	-40	-40	-40	-40
Tensión de Alimentación Máx.	5.5	5.25	5.5	5.25
Tensión de Alimentación Mín.	3	1.8	1.71	4.4
Tipo de Empaquetado	Individual	Cinta Precortada	Individual	Cinta Precortada
Tipo de Puente	USB a UART	USB a UART	USB a UART	USB a UART

En las especificaciones más básicas, se puede apreciar que tanto los componentes de EXAR y MICROCHIP tienen un una tensión de alimentación mínima un tanto elevada para las especificaciones de este proyecto. Aún así, se van mostrar los resultados de investigar con más detalle en los *data sheet* de los componentes.

El primer componente que es descartado es el XR21B1421IL28-F de EXAR ya que no parece tener un *baud rate* configurable de manera sencilla, algo que en el caso de la aplicación desarrollada es esencial, sino que hay que configurarla manualmente mediante el uso de distintas señales. Además, este componente sólo está disponible en encapsulado QFN, lo que hace extremadamente difícil su soldadura a mano.

Los otros 3 componentes sí parecen tener un *baud rate* más intuitivo de configurar, aunque el componente de MICROCHIP no indica en su *data sheet* un ejemplo de configuración del esquemático para construir en la placa, por lo que conllevaría tiempo adicional el diseñar dicho esquemático, y sería más complicado. Lo mismo ocurría con el componente de EXAR, que no disponía de un ejemplo de funcionamiento.

Tanto el FT232RL como el CY7C65213-32LTXI sí poseen en sus *data sheet* ejemplos bastante buenos de posibles configuraciones de esquemáticos. Ambos además tienen un reloj totalmente integrado, y no necesitan ningún cristal externo. El componente MCP2200-I/SO no tenía esta opción por ejemplo, aunque sí el XR21B1421IL28-F, pero ya hemos explicado porque hemos descartado este componente.

Una vez llegados a este punto, se pada asegurar que el FT232RL o el CY7C65213-32LTXI con los candidatos que más requerimientos cubren. Aunque existen aspectos clave para ver en qué se diferencian y el porqué elegir el FT232RL.

Ambos dispositivos poseen unas especificaciones bastante positivas, y se adaptan a los requerimientos de este proyecto. El problema es que para el caso del CY7C65213-32LTXI, tampoco se dispone de un encapsulado distinto a QFN, por lo que resultaría muy complicado soldarlo a mano, como es el caso de este TFG.

Es por tanto que se ha llegado a la conclusión de que la mejor opción es el FT232R, ya que se puede encontrar en encapsulado SSOP, y además dispone de las siguientes funcionalidades que pueden resultar bastante útiles:

- Pines de entrada y salida configurables, *CBUS*, que permiten configurar los LEDS por ejemplo, entre otras posibles aplicaciones potenciales.
- Soporte de interfaces síncronas y asíncronas.
- Alimentación (AVCC: Adc Vcc) integrada, sin necesidad de utilizar filtro externo, mejorando así la integridad de la señal.

Sólo faltaría indicar que el FT2323R es el más caro de los dispositivos analizados, pero a pesar de ello es el más conveniente para utilizar.
A continuación se explicará más en detalle las características del FT232R.

5.3. Dispositivo FT232R

Hoy día la mayoría de ordenadores disponen de conexiones USB para conectar periféricos. Ante la necesidad de tener que llevar la información contenida en la FPGA hacia el ordenador, se vio que, el que dicha información pudiese llegar por USB sería muy beneficioso, ya que se obtendría una aplicación bastante universal.

Siguiendo esta premisa, se pensó en el dispositivo FT232R como se ha explicado en la sección anterior, el cual es una interfaz USB hacia UART serie, y que permite llevar los datos de la UART de la FPGA al ordenador para su posterior análisis. Implementar la pila de comunicación USB directamente es complejo, por lo que no tendría sentido diseñarla específicamente para un trabajo en concreto. Existen varios chips comerciales *of-the-sell* que realizan esta tarea y los cuales tiene un valor en torno a 5€.

Este dispositivo es de la compañía *Future Technology Devices International Ltd.* (FTDI), y entre cuyas características se puede destacar:

- Soporte de drivers para todas las versiones posteriores a Windows 98, Mac OS 8/9, OS-X y Linux 2.4
- Compatible con USB 2.0 *full speed*.
- Interfaz UART para datos de 7 u 8 bits, con bits de *start* y *stop*.
- Pines I/O CBUS configurables.
- EEPROM de 1024 bits.
- Buffers *FIFO* de transmisión y recepción para aplicaciones con una cantidad elevada de datos transferidos.
- Generador de reloj integrado, este permite definir por *software* diferentes velocidades UART sin realizar ningún cambio en el *hardware*.
- Hay 2 encapsulados disponibles, uno de 32 pines QFN, y otro de 28 pines SSOP. En este proyecto, se ha utilizado la versión de 28 pines SSOP.

5.3.1) Señales utilizadas en el FT232R

El FT232R tiene numerosas señales input/output I/O (entrada/salida en inglés), aunque para el caso de este TFG, sólo han sido necesarias un número limitado de ellas. A continuación se puede ver el esquema de todos los pines disponibles.

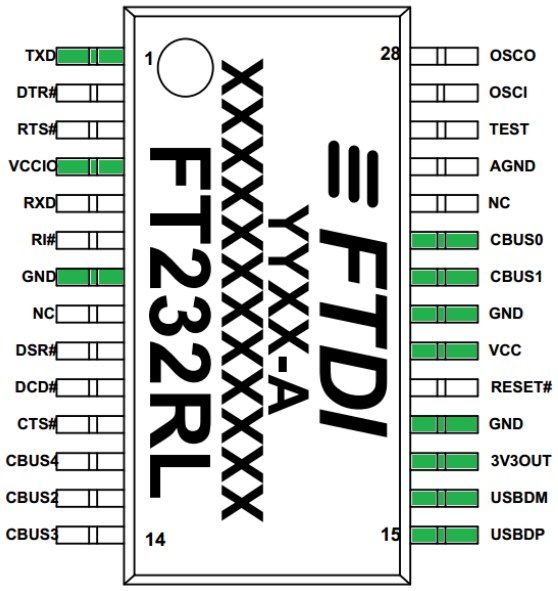

Fig. 47: Pines del FT232R para el encapsulado SSOP. En verde los pines utilizados en el proyecto.

Los pines empleados han sido:

Tabla 3: detalle de los pines utilizados en el dispositivo FT232R.

Pin nº	Nombre	Tipo	Descripción.
15	USBDP	I/O	UDB data *signal plus*. Uno de los 2 pines por donde se transmiten los datos provenientes del USB.
16	USBDM	I/O	USB data *signal minus*.
7, 18, 21	GND	PWR	Conexión a masa.
20	VCC	PWR	Alimentación de +3,3 V hasta de +5,25 V
17	3V3OUT	Output	Salida de +3,3 V que se ha utilizado para alimentar los Leds de CBUS0 y CBUS1.
23	CBUS0	I/O	Por defecto conectada a TXLED#, que es el Led de transmisión. Este Led tiene un pulso activo bajo cuando se transmite data desde el USB al FT232R.
22	CBUS1	I/O	Por defecto conectada a RXLED#, que es el Led de recepción. Este Led tiene un pulso activo bajo cuando se recibe data al USB desde el FT232R.
30	TXD	Output	Salida de transmisión de datos asíncrona.
2	RXD	Input	Entrada de recepción de datos asíncrona.

El resto de pines que se pueden ver en el encapsulado, se han mantenido inactivos.

5.3.2) Diseño del esquemático del FT232R

El esquemático que se ha seguido para su diseño en Altium y su posterior construcción se basa en la diferente documentación que provee el fabricante, en donde se dispone de la información de todos los dispositivos necesarios para poder transmitir la información de la FPGA al ordenador, y viceversa.

También se añade el esquema de conexiones de los Leds con los CBUS, para que se pueda ver visualmente si el dispositivo se encuentra transmitiendo y/o recibiendo información.

Basándose por tanto en lo anteriormente dicho, se ha llevado a cabo el siguiente esquemático en *Altium Designer*:

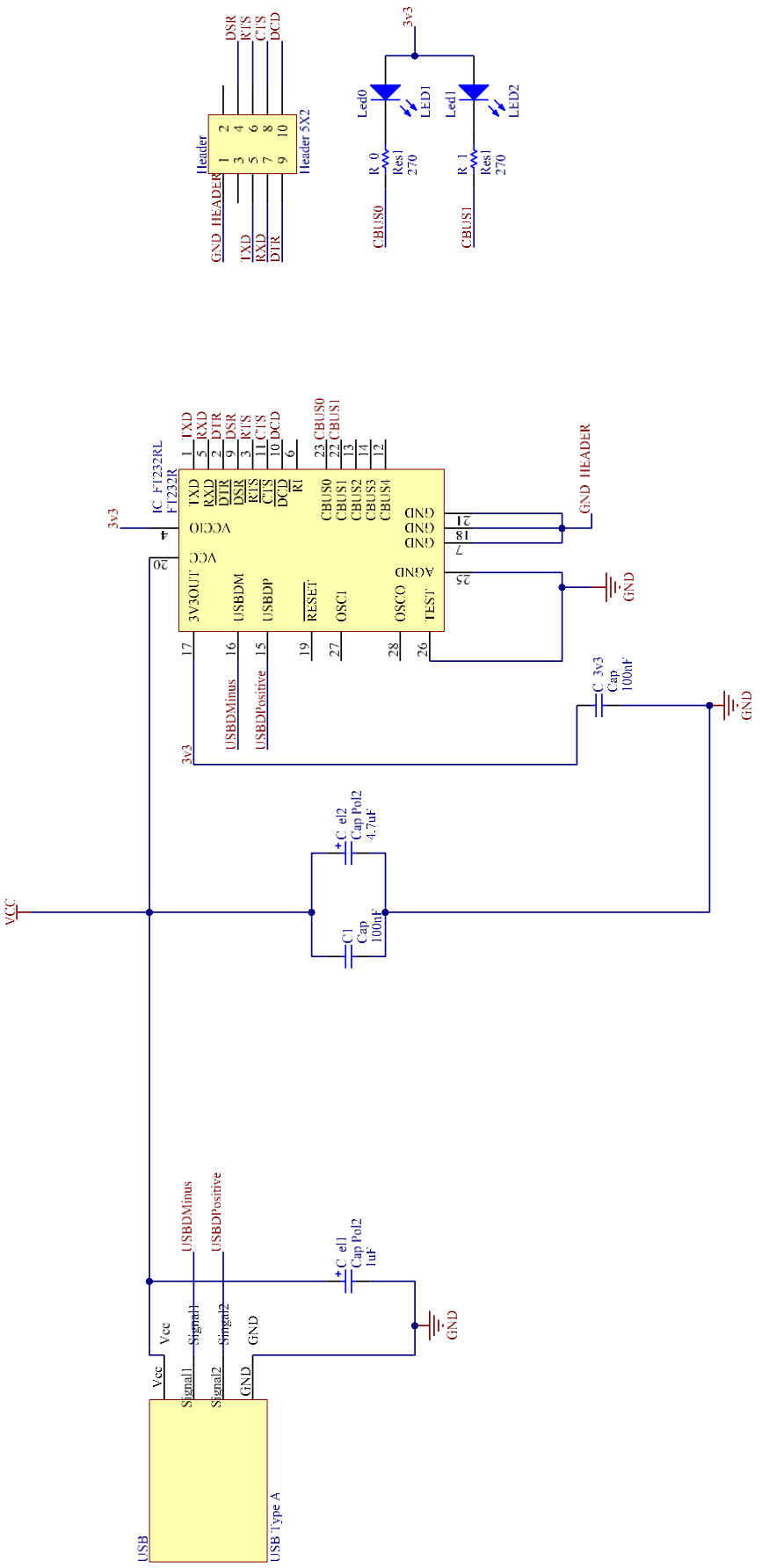

Fig. 48: Esquemático en Altium del diseño USB a MCU UART.

5.3.3) Diseño del circuito impreso del FT232R

Tras el diseño del esquemático en la sección anterior, se ha creado el circuito impreso de doble cara, utilizando para ello *Altium Designer*.

Al utilizar componentes *Sourface-Mount Devices* (SMD) el rutado de las señales se ha hecho principalmente en la capa superior, *top layer*. A pesar de ello, y puesto que las dimensiones de esta placa son reducidas, se ha necesitado de una capa inferior, *bottom layer*, por donde seguir rutando las señales de la capa superior gracias a vías en el PCB.

A continuación se muestran 3 figuras, la figura 49 en donde se puede observar la *top layer* del PCB, la figura 50, donde se observa la *bottom layer*, y la figura 51, donde ambas capas están superpuestas.

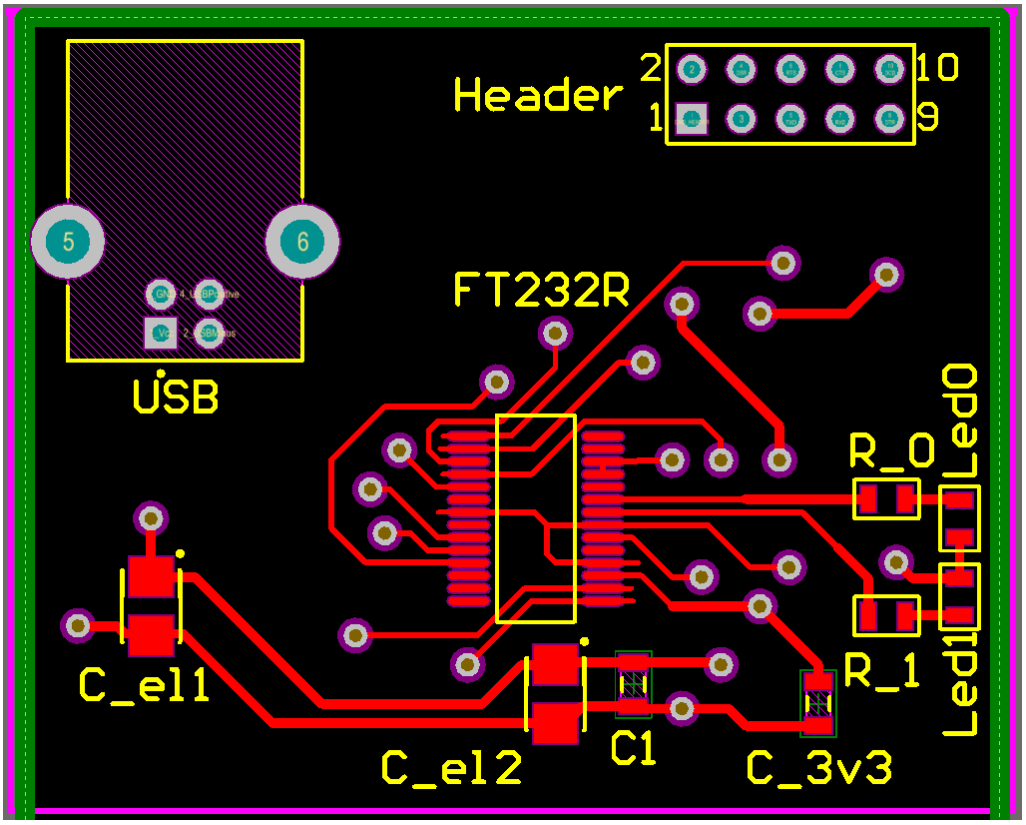

Fig. 49: Capa superior del PCB diseñado en Altium Designer.

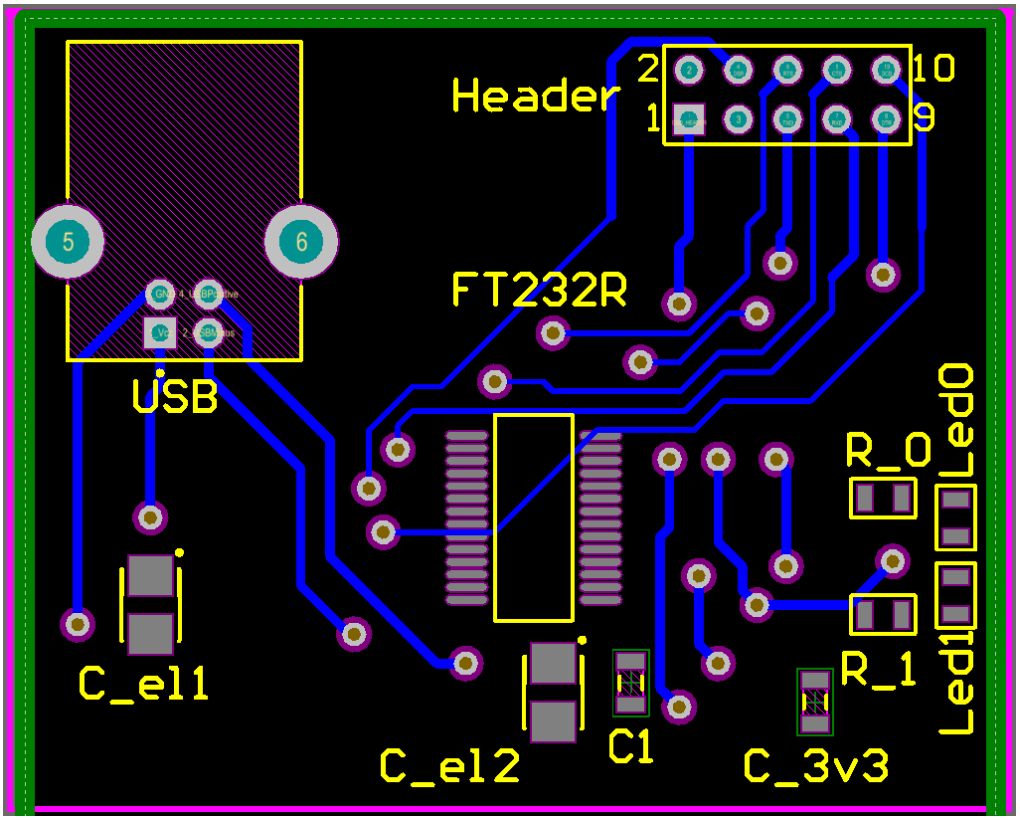

Fig. 50: Capa inferior del PCB diseñado en Altium Designer.

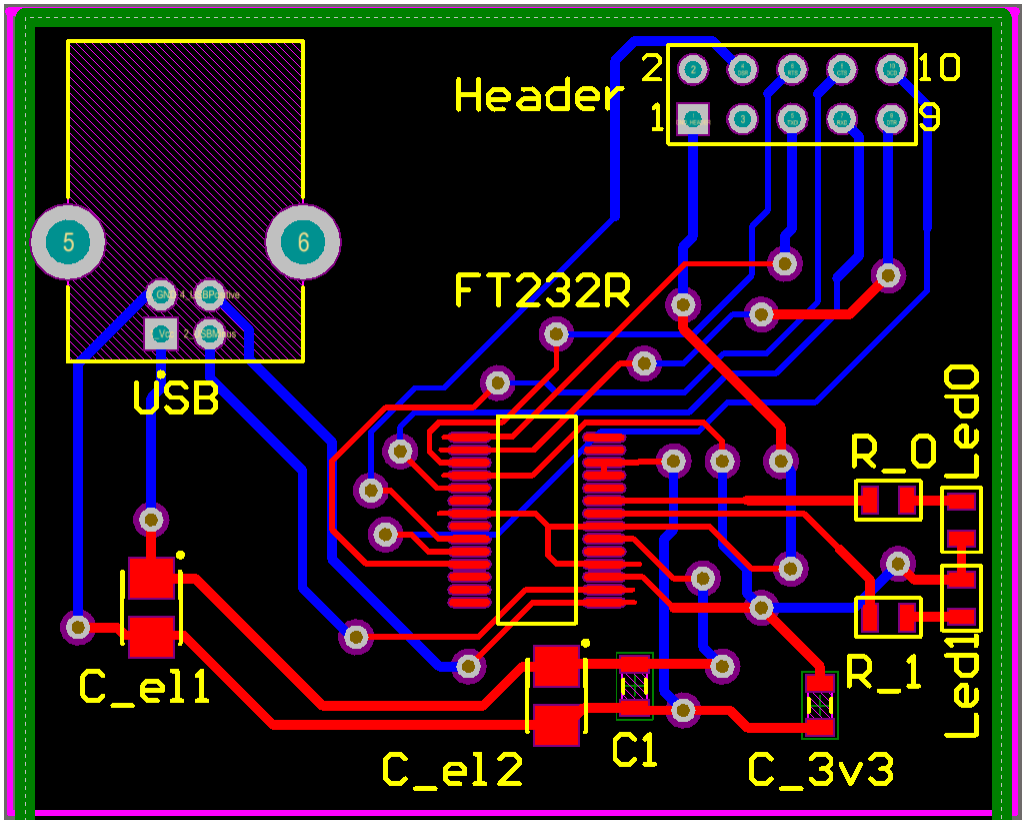

Fig. 51: Capas superior e inferior superpuestas del PCB del diseñado en Altium Designer.

Para el diseño de este PCB, es necesario aportar a *Altium Designer* de las librerías con las distintas huellas (*footprints* por su terminología en inglés*)* de los componentes. En el caso de resistencias, leds, USB tipo B, conector *header* y condensadores, las librerías se pueden obtener de la misma página web de Altium. Por otro lado, para el caso del FT232R, los *footprints* disponibles para los distintos encapsulados pueden descargarse de la página web del fabricante de manera sencilla.

Con respecto al PCB, habría que hacer unos comentarios sobre el procedimiento que se ha seguido, así como el detalle de los componentes utilizados (*bill of materials, BOM*).

Los componentes utilizados han sido:

- 1 FT232R encapsulado SSOP. Precio aproximado: 3,52 €.
- 1 USB tipo B de 12 mm x 10,8 mm. Precio aproximado: 0,42 €
- 1 conector *header* 5x2. Precio aproximado: 0,35 €
- C_el1: condensador electrolítico de tántalo de clase A. Valor 1 µF. Clase EIA 3216-18. Precio aproximado: 0,30 €.
- C_el2: condensador electrolítico de tántalo de clase B. Valor 4,7 µF. Clase EIS 3528-21. Precio aproximado: 0,27 €.
- C1 y C_3v3: condensadores SMD0805 de 100 nF. Precio aproximado: 0,10 € cada uno.
- R0 y R1: resistencias SMD0805. Valor: 270 **Ω**. Precio aproximado: 0,01 € cada uno.
- Led0 y Led1: Leds SMD0805. Precio aproximado: 0,25 € cada uno.

La fresadora utilizada ha sido de la compañía LPKF, modelo ProtoMat S100, y el proceso de soldadura se ha realizado manualmente sin el uso de ninguna herramienta automática.

El precio final del diseño serían unos 5,35 €, si a eso añadimos unos 2 € de la placa de cobre, tenemos un precio total aproximado de 7,35 €, un PCB bastante barato.

Durante el diseño, se han tenido que seguir ciertos criterios para poder soldar e imprimir de manera correcta la placa.

En primer lugar, como se comenta en párrafos anteriores, puesto que los componentes utilizados en su mayoría son SMD, se ha necesitado hacer vías para rutar las señales por la *bottom layer*, BL (pistas en azul), además de hacer las correspondientes pistas en la *top layer*, TL (pistas en rojo). En estas vías se ha incrementado la *clearance* que por defecto Altium establece en unos 0,1 mm hasta unos 0,4 mm, ya que a la hora de soldar en un primer intento de construcción de la placa, resultó ser insuficiente para evitar cruce indeseado de señales a la hora de soldar a mano.

Asimismo, también se ha incrementado la anchura por defecto de las pistas hasta los 0,5 mm en las proximidades de los pines a soldar. Esta anchura se ha mantenido en

0,3 mm para las pistas que no hacía falta soldar pines. Se ha evitado también el tener pistas con ángulos de 90º, ya que esto podría causar problemas a altas frecuencias.

En las 2 figuras que se muestran a continuación se pueden observar la capa superior y la capa inferior del circuito impreso diseñado. Dicho circuito impreso posee unas dimensiones de 4 cm de ancho por 5 cm de largo.

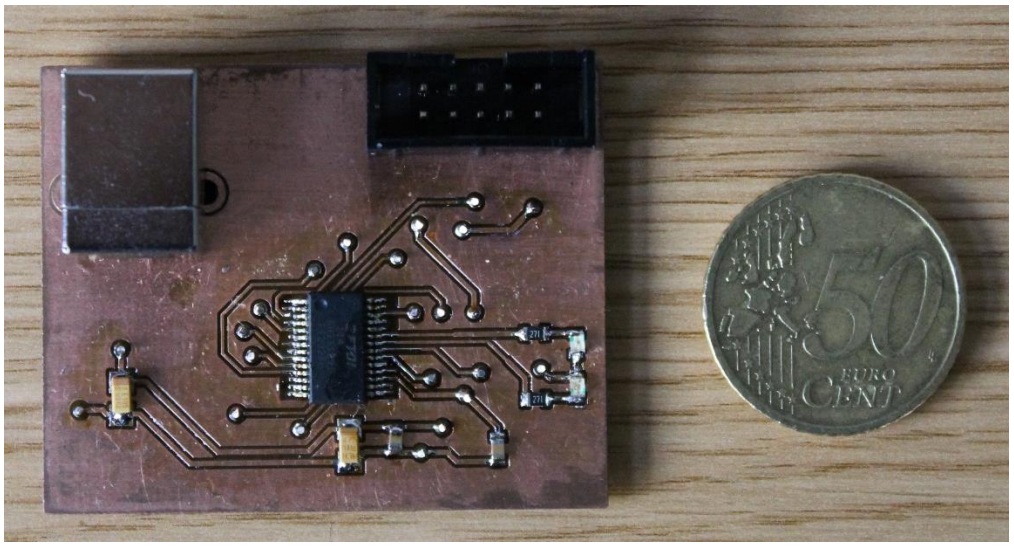

Fig. 52: Capa superior del PCB construido.

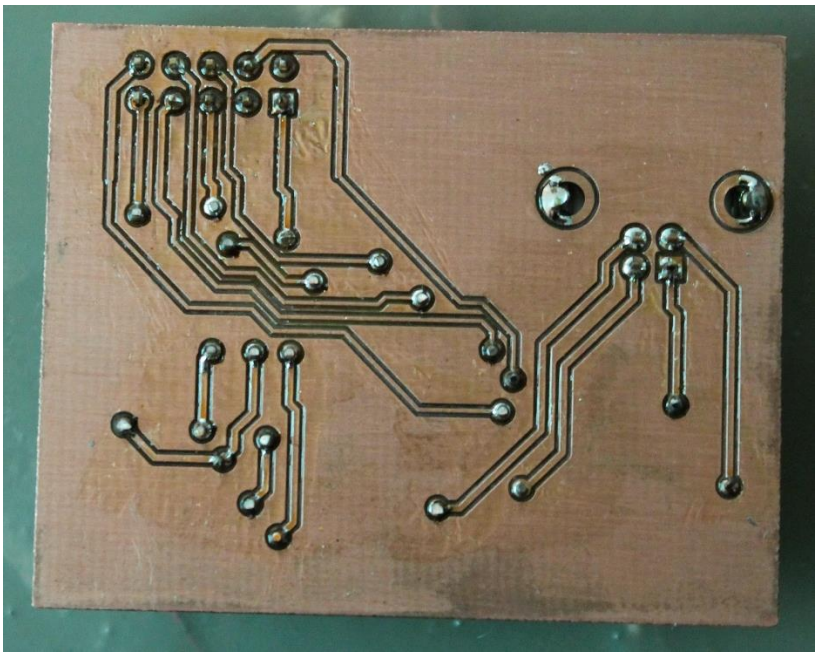

Fig. 53: Capa inferior del PCB construido.

5.4. Interconexión UART/FPGA

Como se puede observar en el diseño del PCB, la placa diseñada tiene 10 conexiones de salida hacia el ordenador. Estas señales provienen del chip FT232R.

A pesar de ello, en principio sólo se necesitan tres de ellas para el funcionamiento de este TFG: GND (conexión a masa), TXD (transmisión de datos UART) y RXD (recepción de datos UART), aunque el poner el resto de señales puede tener una potencial utilidad en hipotéticos desarrollos de este TFG. Un ejemplo serían las señales RTS/CLS, que se han adjuntado para que el PCB de conversión sea compatible con las actualizaciones mencionadas, o incluso con otros proyectos que necesiten de conversión UART/USB.

Un apunte importante es que la señal de recepción en el USB-UART, en este caso RXD, tendrá que estar conectada a la señal de salida de datos en la FPGA, por la que se están mandando los bytes de las tramas. A su vez, el pin de transmisión de la placa USB-UART tendrá que estar conectada a la señal de recepción en la FPGA, por si la interfaz gráfica en C# estuviese interesada en enviar algún dato.

La placa de control del convertidor tiene 3 conectores de I/O, denominados A1, B1 y A2. En este caso se ha utilizado el conector A1.

La configuración y distribución de los pines de dicha placa pueden observarse en la hoja de datos del fabricante [13].

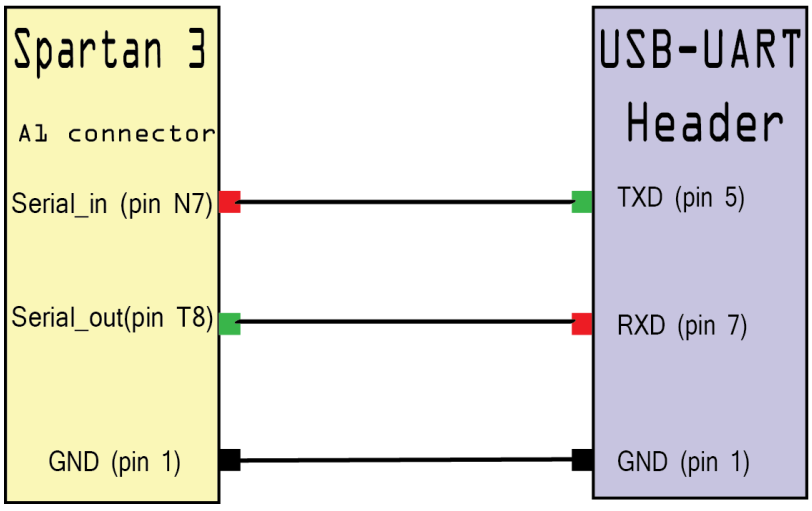

Fig. 54: Esquema de conexiones entre el conector A1 de la placa de control del convertidor y la placa USB-UART.

6. MÓDULO DE LA INTERFAZ EN C#

El módulo de simulación engloba todos aquellos elementos que hacen posible la representación e interpretación de los datos extraídos del convertidor de potencia conmutado mediante la aplicación desarrollada por *software*.

6.1. Introducción: lenguaje de programación C#

En esta Trabajo Fin de Grado se ha desarrollado una aplicación en lenguaje C *Sharp*, C#, utilizando *software* y bibliotecas de Microsoft. El entorno de desarrollo integrado (IDE por sus siglas en inglés) utilizado ha sido el Visual Studio Express 2013, en el cual se ha elaborado un proyecto del tipo *Windows form application*.

C# es un lenguaje de programación orientado a objetos desarrollado y estandarizado por el fabricante Microsoft, cuyo objetivo es aumentar la productividad del programador con respecto a otros lenguajes de programación. En los lenguajes de programación orientados a objetos, un programa consiste en numerosos objetos, *objects*, que interaccionan entre sí por medio de acciones. Estas acciones que un objeto puede tomar se denominan métodos. Todas las funciones se consideran *métodos*.

C#, a pesar de tener una sintaxis basada en C y C++, tiene sus propias peculiaridades, como por ejemplo el incluir encapsulamiento, herencia y polimorfismo [15]. Un encapsulamiento significa crear una frontera alrededor del objeto, para separar su comportamiento externo (*public*) de sus detalles de implementación internos (*private*).

Los aspectos claves que hacen a C# tan potente es que trabaja con tipos, *types,* los cuales comparten el mismo *set* básico de funcionalidades gracias a su *unified type system*. Por ejemplo, la instancia de cualquier tipo de dato puede convertirse a una cadena de caracteres (*string*) simplemente utilizando el método `ToString` [16].

En C# a su vez se trabaja con clases e interfaces, a diferencia de otros lenguajes de programación orientados a objetos que sólo utilizan clases. Una clase es una construcción que permite crear *types* personalizados mediante la agrupación de variables de otros tipos, métodos y eventos. Una clase es como una copia heliográfica, define los datos y el comportamiento de un *type*. Una interfaz contiene las definiciones de un grupo de funciones relacionadas que una clase o *struct* puede implementar. Las interfaces son especialmente útiles en escenarios donde se necesitan numerosas "herencias".

Otras características que se encuentran en C# son las *propiedades* y los *eventos*. Una propiedad es un miembro que ofrece un mecanismo flexible para leer, escribir o calcular el valor de un campo privado (*private field*). Un evento en C# es la forma que tiene una clase de proveer notificaciones a los clientes de esa clase, cuando le ocurre algo interesante a un objeto con el que interaccionan [17].

6.2. Interfaz diseñada en C#

A continuación, se va a proceder a introducir la interfaz gráfica final desarrollada, *Graphical User Interface*, GUI por sus siglas en inglés, así como una explicación de las partes que componen la misma y como se ha programado. Se puede observar el resultado final de dicha interfaz en la figura 55.

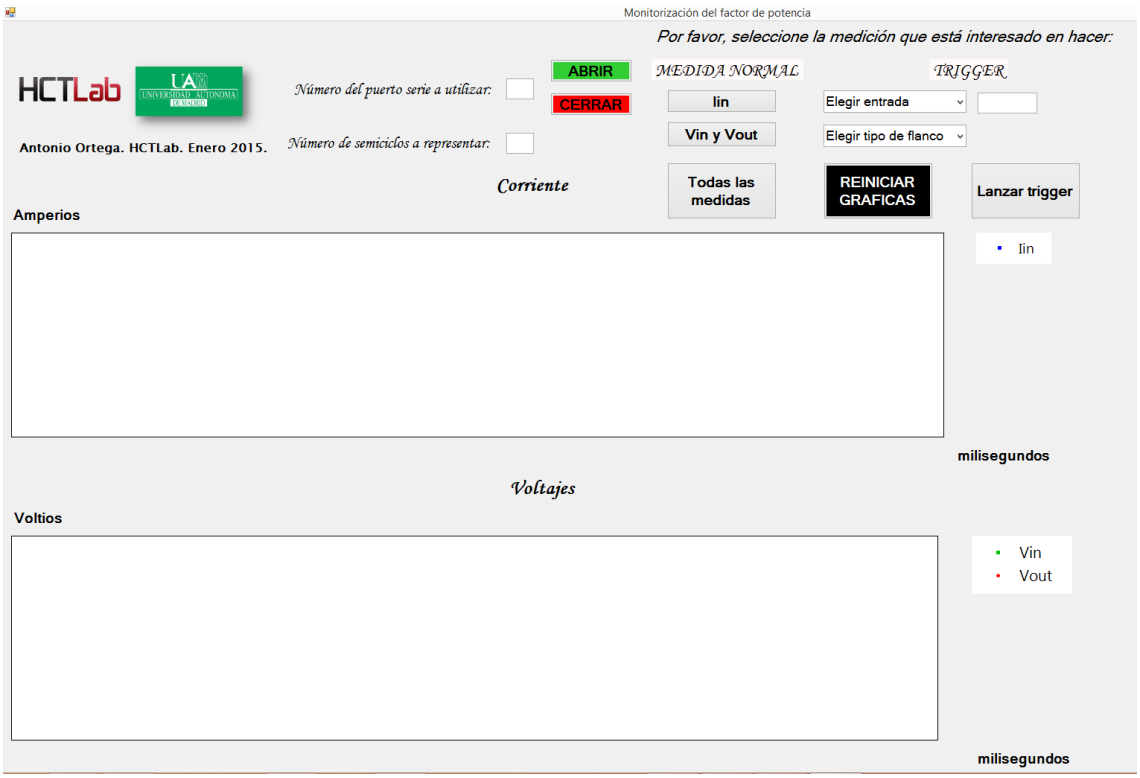

Fig. 55: Interfaz gráfica de la aplicación desarrollada en C#.

Esta GUI tiene la intención de actuar a modo de *osciloscopio* digital, de cara al análisis de los datos que recibe por el puerto serie. Es por ello que posee diferentes funcionalidades integradas, como puede ser la opción de seleccionar un *trigger*, el número de semiciclos a representar…

Como se puede observar, la interfaz está dividida principalmente en 2 zonas: la selección de los datos a representar (situado en la parte superior) y la representación de los datos mediante gráficas (parte medio-inferior).

A la hora de seleccionar los datos, el usuario tiene diversas opciones. En primer lugar debe decidir si en la medida que va a realizar quiere utilizar un *trigger* o no. Si el usuario decide tomar una *captura inmediata*, podrá elegir si quiere visualizar los datos recibidos de corriente de entrada y tensiones de entrada y salida, o si por el contrario prefiere visualizar sólo la corriente de entrada, o sólo las tensiones. A su vez, el usuario debe seleccionar el número de semiciclos que está interesado en examinar, así como abrir el puerto de comunicación serie en el que se encuentre conectado el dispositivo USB. El usuario también podrá cerrar dicho puerto serie cuando cese de utilizar la interfaz.

Si una vez terminada esta primera medida sin *trigger*, el usuario está interesado en seguir tomando medidas, puede pulsar el botón de *reiniciar gráficas*, el cual no sólo limpiará todos los puntos que en ese momento se encuentren representados en las distintas gráficas de corriente y voltajes, sino que también borrará toda la información contenida hasta el momento tanto en los buffer del puerto serie, como en las listas dinámicas de la aplicación. De esta manera, el usuario podrá retomar la medición de valores sin necesidad de reiniciar el programa.

En segundo lugar, el usuario puede decidir lanzar un *trigger* en su medida. El menú configurable de esta modalidad se encuentra bajo la etiqueta titulada *TRIGGER*. Aquí el usuario también tiene capacidad de decisión, por una parte tiene que decidir sobre qué datos quiere lanzar este *trigger*, Iin para referirse a la corriente de entrada, Vin para referirse a la tensión de entrada, o Vout para referirse a la tensión de salida. Una vez seleccionado sobre qué datos quiere realizar el trigger, el usuario debe escoger si está interesado en realizar un *trigger* con flanco positivo o negativo. Para elegir el valor numérico del trigger, éste se introducirá manualmente en el cuadro de texto.

Si el flanco del *trigger* es positivo: cuando se reciba una medida por debajo del valor introducido, y la siguiente por encima, entonces se representarán el número de semiciclos que el usuario desee cogiendo los datos a partir de ese punto. Si el flanco es negativo, sería el revés: cuando se reciba una medida por encima del *trigger*, y la siguiente por debajo, se representan los semiciclos. Una vez configurado el *trigger*, se tienen unas etiquetas en la parte superior derecha en las que se muestra la información seleccionada: se mostrará el tipo de dato seleccionado, así como su valor numérico, su unidad de medida, y su tipo de flanco.

Del mismo modo que para el caso de la medida normal, el usuario puede decidir reiniciar las gráficas para seguir tomando medidas, sin necesidad de reiniciar el programa.

A la hora de representar los datos en las gráficas se ha intentado hacer con la mayor exactitud posible, es por ello que se necesita el mayor número de puntos por semiciclo, pero sin llegar a saturar el buffer de transmisión. En el código VHDL se vio que se construye una trama por cada 10.000 ciclos de reloj. Como la frecuencia de reloj del convertidor es de 100 Mhz, esto significa que se está mandando:

$$\frac{100 * 10^6}{10.000} = 10.000\ puntos/segundo$$

Y como un semiciclo de red eléctrica europea tiene una duración aproximada de 10 ms, o lo que es lo mismo, tiene una frecuencia de 100 Hz:

$$\frac{10.000\ puntos/segundo}{100\ Hz} = 100\ puntos$$

Se consiguen por tanto 100 puntos por cada semiciclo representado en las gráficas. Estos 100 puntos debieran ser suficientes para apreciar la forma del semiciclo sin problema. Además, el eje X de las gráficas es auto-escalable, lo que quiere decir que

cuantos menos semiciclos se representen, mayor distancia habrá entre los puntos, y viceversa.

6.2.1) Interfaz diseñada en C#: *etiquetas*

En primer lugar, habría que señalar que se han utilizado diversas etiquetas, *labels*, sobre los que escribir los distintos textos que se pueden encontrar en la interfaz gráfica. En la figura 56 se puede encontrar la definición en C# de uno de los *label* utilizados, y en la figura 57 su correspondiente visualización en la interfaz gráfica.

```
private System.Windows.Forms.Label label3;
this.label3 = new System.Windows.Forms.Label();
this.label3.AutoSize = true;
this.label3.BackColor = System.Drawing.Color.Snow;
this.label3.Font = new System.Drawing.Font("Monotype Corsiva", 18F, System.Drawing.FontStyle.Italic,
                                           System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.label3.Location = new System.Drawing.Point(1161, 63);
this.label3.Margin = new System.Windows.Forms.Padding(4, 0, 4, 0);
this.label3.Name = "label3";
this.label3.Size = new System.Drawing.Size(258, 37);
this.label3.TabIndex = 21;
this.label3.Text = "MEDIDA NORMAL";
```

Fig. 56: Código en C# para declarar las propiedades de una etiqueta.

Fig. 57: Visualización de la etiqueta Medida Normal *del código C# anterior.*

Como se puede observar, todas las funciones se encuentran contenidas en la biblioteca *system* de Microsoft.

La utilidad de las etiquetas reside en que el texto que muestran se puede sustituir de manera sencilla accediendo a la propiedad `Text` de la misma. Es así como se consiguen representar unas etiquetas u otras para el caso de la medida con *trigger*.

6.2.2) Interfaz diseñada en C#: *botones*

Por otro lado, se ha necesitado disponer de *botones* en la interfaz gráfica. El objetivo principal de estos botones es llevar a cabo unas acciones determinadas u otras tras pulsarlos en el código principal del programa. De la misma manera que para el caso de las etiquetas, los botones pueden contener texto de distintos formatos o no, el cual puede modificarse, aunque esta no es su principal utilidad.

Para ilustrar el funcionamiento de los botones, a continuación se muestran 3 figuras de ejemplo. La figura 58 contiene el código en C# para la declaración del botón *Todas las medidas*, el cual tras pulsarlo, mostrará los semiciclos de corriente y voltajes introducidos por el usuario, obteniendo los datos del puerto serie que el usuario a su vez ha seleccionado. La figura 59 contiene la representación visual de este botón; y por último la figura 60 muestra la función que es llamada en el código principal del programa tras pulsar el botón. Como se puede observar, tras pulsar el

botón *Todas las medidas*, se activarán diversos *flags* de control que hay en el código, y después se llamará a la función de LecturaDatos(), la cual lee los datos por el puerto serie y los representa, en función a las características de la medida a tomar.

```
private System.Windows.Forms.Button button3_todos;
this.button3_todos = new System.Windows.Forms.Button();
this.button3_todos.BackColor = System.Drawing.Color.Transparent;
this.button3_todos.Font = new System.Drawing.Font("Microsoft Sans Serif", 13.8F, System.Drawing.FontStyle.Bold,
                                                   System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.button3_todos.ForeColor = System.Drawing.SystemColors.ActiveCaptionText;
this.button3_todos.Location = new System.Drawing.Point(1189, 234);
this.button3_todos.Margin = new System.Windows.Forms.Padding(3, 2, 3, 2);
this.button3_todos.Name = "button3_todos";
this.button3_todos.Size = new System.Drawing.Size(196, 94);
this.button3_todos.TabIndex = 8;
this.button3_todos.Text = "Todas las medidas";
this.button3_todos.UseVisualStyleBackColor = false;
this.button3_todos.Click += new System.EventHandler(this.button3_todos_Click);
```

Fig. 58: Código en C# para la declaración de las propiedades de un botón.

Fig. 59: Visualización del botón Todas las Medidas *del código C# anterior.*

```
private void button3_todos_Click(object sender, EventArgs e)
{
    VariablesGlobales.botonPulsado = 1;
    VariablesGlobales.MedirTodo = 1;
    LecturaDatos();
}
```

Fig. 60: Función que es llamada tras pulsar el botón de la figura anterior.

6.2.3) Interfaz diseñada en C#: *cuadros de texto*

Los cuadros de texto son de gran utilidad en la interfaz en C# desarrollada, puesto que son la vía mediante la cual el usuario puede introducir diversos valores numéricos que configuran la medida que va a tomar.

Al igual que el resto de elementos, los cuadros de texto deben declararse e inicializarse. La particularidad que presentan estos cuadros de texto es que, una vez declarados, en cualquier punto del código del programa se podrá acceder a su contenido simplemente llamando a la propiedad `Text` del botón: `textBox_Nombre.Text`, en donde se contendrá el valor o valores en forma de cadena de caracteres. La conversión de cadena de caracteres a valor numérico *int* es extremadamente sencillo en C#, simplemente invocando a la función `Convert.ToInt16(textBox_Nombre.Text)`.

La figura 61 a continuación muestra un ejemplo de configuración de un cuadro de texto, en este caso el cuadro de texto en el que el usuario introduce el número de semiciclos que quiere ver representados.

En la figura 62 se observa también cómo obtener el número contenido en el cuadro de texto.

```
private System.Windows.Forms.TextBox textBox_senos;

this.textBox_senos = new System.Windows.Forms.TextBox();
this.textBox_senos.Font = new System.Drawing.Font("Microsoft Sans Serif", 13.8F, System.Drawing.FontStyle.Regular,
                                                  System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.textBox_senos.Location = new System.Drawing.Point(908, 185);
this.textBox_senos.Margin = new System.Windows.Forms.Padding(4);
this.textBox_senos.Name = "textBox_senos";
this.textBox_senos.Size = new System.Drawing.Size(48, 34);
this.textBox_senos.TabIndex = 24;
this.textBox_senos.TextChanged += new System.EventHandler(this.textBox1_TextChanged_1);
```

Fig. 61: Código en C# para la declaración de las propiedades de un cuadro de texto.

```
string numeroSenos = textBox_senos.Text;
try
{
    VariablesGlobales.NumeroSenosArepresentar = Convert.ToInt16(numeroSenos);
}
catch (Exception ex)
{
    MessageBox.Show("No se ha especificado un número de semiciclos. Intentelo de nuevo.");
    return;
}
```

Fig. 62: Código para la obtención del número de semiciclos del cuadro de texto de la figura anterior.

6.2.4) Interfaz diseñada en C#: *gráficos*

Los gráficos son la pieza fundamental de la interfaz gráfica desarrollada, puesto que llevan a cabo la labor más importante: la de representar los datos. Como se puede apreciar en la figura 63, la declaración de un gráfico en C# se hace de la misma forma que para declarar un botón, una etiqueta o un cuadro de texto. Lo que diferencia al gráfico, es que tiene muchas más opciones configurables.

Las *propiedades* y *métodos* básicos utilizados del *Class* de tipo *Chart* en C#, son [18]:

`public ChartAreaCollection ChartAreas {get;}`	Obtiene el objeto `ChartArea` (configuración del aspecto visual del gráfico).
`public int Height {get; set;}`	Obtiene o establece la altura del gráfico.
`public string Name {get; set;}`	Obtiene o establece el nombre del gráfico.
`public SeriesCollection Series {get;}`	Obtiene el objeto `Series` (los puntos a representar).
`public TitleCollection Titles {get;}`	Obtiene o establece el objeto `Titles` (títulos del gráfico).

Tabla 4: Propiedades y métodos básicos en el Class *Chart* en C#.

Puesto que el código de configuración de un gráfico es bastante extenso, este se adjuntará en el anexo II, aunque a continuación se van a resumir las partes más importantes a configurar en un gráfico:

- ***Series***: son la lista de puntos a visualizar. Como para el ejemplo del gráfico que representa los voltajes de la figura 65 tenemos por un lado el voltaje de entrada y por otro el voltaje de salida, habrá 2 series en este gráfico. Estas *series* pueden ser configuradas tanto en el color que van a tener, así como el tamaño, el grosor, el nombre a visualizar en la gráfica …
  En la figura 64 se puede observar cómo ir añadiendo los valores de voltaje a la lista de puntos. Se debe especificar tanto la posición X como la posición Y del punto para que Windows sepa representarlo.
- ***Legend:*** leyendas. En la figura 65 se pueden observar en la parte derecha. Estas leyendas pueden ser configuradas visualmente de la manera que se desee.
- ***ChartArea:*** área del gráfico. Para esta área puede configurarse su tamaño, distribución, valor máximo en los ejes X/Y …

- ***Tittles:*** Títulos del gráfico. Son útiles para poder indicar por ejemplo el nombre del gráfico, las unidades de medidas en los ejes X e Y, u otra información relevante que se quiere mostrar al usuario.

Puesto que las gráficas están representando semiciclos de corrientes y voltajes, el eje X debe llevar asociada una medida de tiempo. Como Windows no sabe cuántos milisegundos tiene cada semiciclo, ha de introducirse manualmente los valores en el código principal usando para ello la opción de *CustomLabel* que permite al programador introducir etiquetas personalizadas en cualquier gráfica del proyecto.

```
this.chart4 = new System.Windows.Forms.DataVisualization.Charting.Chart();
```

Fig. 63: Declaración del gráfico que representará los voltajes de entrada y salida.

```
else if (VariablesGlobales.MedirVoltajes == 1)
{
    VariablesGlobales.V_in.Add(V_in);
    Indice_V_in = VariablesGlobales.V_in.Count;
    chart4.Series[0].Points.AddXY(Indice_V_in, V_in);

    VariablesGlobales.V_out.Add(V_out);
    Indice_V_out = VariablesGlobales.V_out.Count;
    chart4.Series[1].Points.AddXY(Indice_V_out, V_out);

}
```

Fig. 64: Introducción de puntos en el gráfico declarado en la figura anterior.

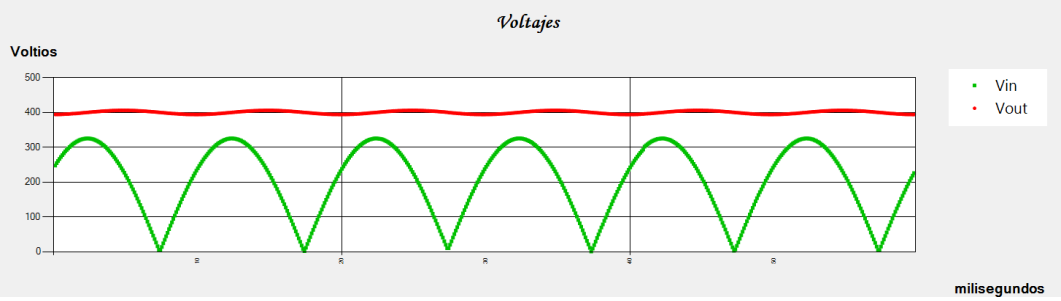

Fig. 65: Ejemplo de captura de datos de voltajes que han sido representados en el gráfico.

El crear las etiquetas personalizadas que hagan referencia al tiempo es sencillo, puesto que, como de explicó anteriormente, cada semiciclo es representado mediante 100 puntos. Tan sólo haría falta incrementar 10 ms cada 100 puntos representados en el eje X. Esto se puede observar en la figura 66. Desafortunadamente, Windows no permite variar el tamaño del texto introducido, de ahí a que dichos números salgan más pequeños.

```
int contadorLabel = 0;
while (contadorLabel < 50)
{
    string leyenda = Convert.ToString(contadorLabel * 10);
    chart3.ChartAreas[0].AxisX.CustomLabels.Add(contadorLabel * 90, contadorLabel * 110, leyenda);
    chart4.ChartAreas[0].AxisX.CustomLabels.Add(contadorLabel * 90, contadorLabel * 110, leyenda);

    contadorLabel++;
}
```

Fig. 66: Introducción de etiquetas personalizados en los ejes X de las gráficas en C#.

6.2.5) Interfaz diseñada en C#: *puerto serie*

El puerto serie es otro de los aspectos clave del proyecto desarrollado en C#, ya que es a través del cual se reciben los bits que contienen la información del convertidor de potencia *boost*.

Las *propiedades* y *métodos* básicos utilizados del *Class* de tipo *SerialPort* en C#, son [18]:

`public void Close()`	Cierra el puerto serie.
`public void Open()`	Abre el puerto serie.
`public int BaudRate {get; set;}`	Establece o lee el *baud rate*.
`public int BytesToRead {get;}`	Obtiene el número de bytes a leer en el buffer de entrada.
`public bool IsOpen {get;}`	Obtiene si el puerto serie se encuentra abierto o cerrado.
`public Parity Parity {get; set;}`	Obtiene o selecciona una paridad determinada.
`public string PortName {get; set;}`	Obtiene o escribe el nombre del puerto serie.
`public int ReadBufferSize {get; set;}`	Obtiene o selecciona el tamaño del buffer de entrada del puerto serie.
`public void DiscardInBuffer()`	Limpia el buffer de entrada del puerto serie.
`public int ReadByte()`	Lee un byte síncronamente del buffer de entrada del puerto serie.
`public int ReadChar()`	Lee un caracter síncronamente del buffer de entrada del puerto serie.

Tabla 5: Propiedades y métodos básicos en el Class *SerialPort* en C#.

Configurar el puerto serie en un proyecto de *Windows form application* es extremadamente, sencillo, y puede apreciarse en la figura 67. Los únicos cuatro datos que el programador debe indicar al entorno de desarrollo es:

- Nombre a utilizar por el puerto serie dentro del programa. En este ejemplo *serialPort1*.
- Velocidad de transmisión: que para el caso de este proyecto era de 3 MBaudios.
- Nombre del puerto serie: que por defecto en este proyecto se establece en el "COM3", pero el usuario como se apreció en la interfaz gráfica puede introducir otro valor manualmente.
- Tamaño del buffer donde se van almacenando los datos recibidos en el puerto serie. Este buffer tiene que ser lo suficientemente grande para que no se pierda ningún dato. La unidad del dato introducido es bytes.

```
private System.IO.Ports.SerialPort serialPort1;
this.serialPort1 = new System.IO.Ports.SerialPort(this.components);
this.serialPort1.BaudRate = 3000000;
this.serialPort1.PortName = "COM3";
this.serialPort1.ReadBufferSize = 65536;
```

Fig. 67: Código en C# para la declaración de un puerto serie.

El resto de configuraciones se mostrarán en los códigos en C# anexos.

En la figura 68 se muestra el algoritmo para leer los datos por el puerto serie, y cómo estos se almacenan en la lista dinámica `VariablesGlobales.ListaBytesLeido()`.

Una vez introducidos los bytes en esta lista dinámica, se procederá a examinarlos para llevar a cabo las acciones que el usuario haya seleccionado en la GUI: representar corriente y/o voltajes, medida con/sin *trigger* …

```
VariablesGlobales.bytesPedidos = VariablesGlobales.NumeroSenosArepresentar * 100 * 7 * 5;
while (bytesLeidos < VariablesGlobales.bytesPedidos)
{
    while (serialPort1.BytesToRead != 0 && bytesLeidos < VariablesGlobales.bytesPedidos)
    {
        // Bucle que se ejecuta mientras haya bytes en el puerto serie para leer.
        VariablesGlobales.BytesLeido = serialPort1.ReadByte();
        VariablesGlobales.ListaBytesLeido.Add(VariablesGlobales.BytesLeido );
        bytesLeidos++;
    }
}
```

Fig. 68: Algoritmo para leer los bytes del puerto serie en C#.

7. RESULTADOS

Una vez expuestos y desarrollados los distintos módulos (comunicación, UART-USB e interfaz gráfica), que componen este TFG, es el momento de mostrar los resultados que se obtienen en la interfaz gráfica al interconectar los módulos tal y como se explicaba en capítulo 3, el cual aportaba un enfoque global del proyecto desarrollado.

Los resultados se mostrarán diferenciados en dos secciones: con y sin *trigger*, ya que son los dos modos de funcionamiento disponibles distintos.

7.1. Resultados de medidas sin *trigger*

Como se comentaba en el capítulo anterior, a la hora de realizar una medida normal, el usuario puede elegir si mostrar los valores de corriente, los valores de voltaje, o ambos.

En el caso de la figura 69, se muestra la representación de un semiciclo de red de la corriente de entrada Iin, y los voltajes de salida Vout y de entrada Vin. Como el eje X es auto escalable, se pueden distinguir claramente los 100 puntos utilizados para representar el semiciclo.

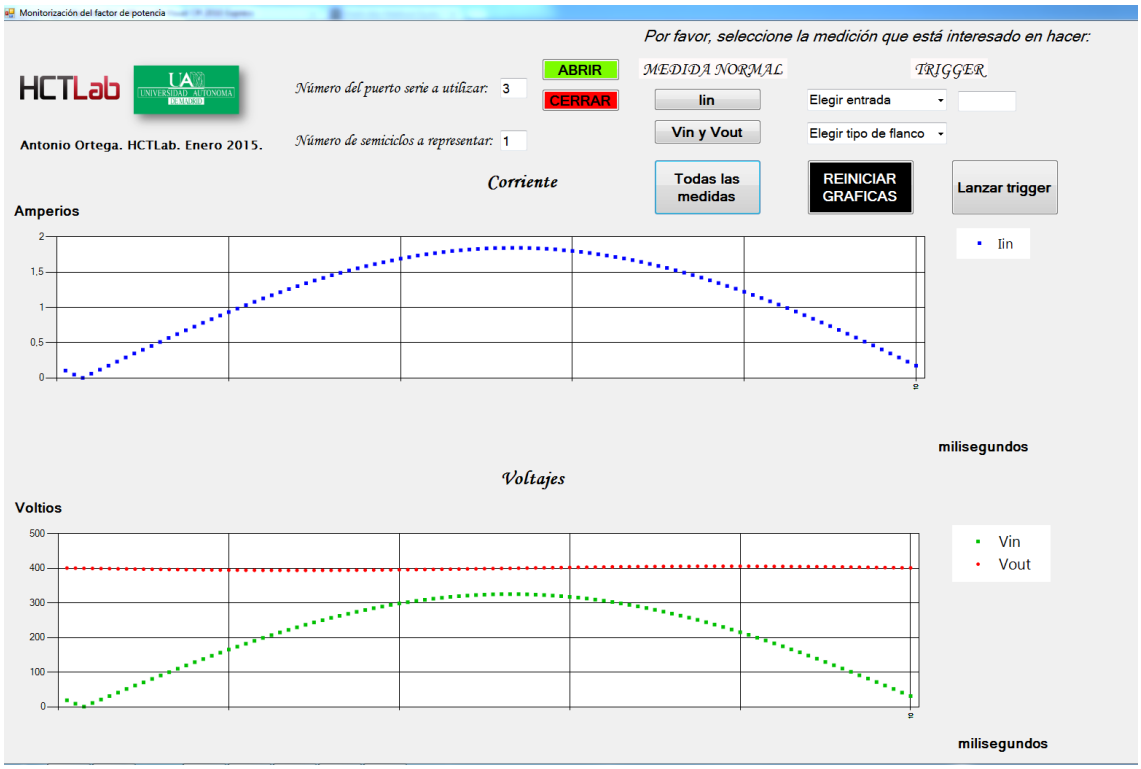

Fig. 69: Gráficas de Iin, Vin *y* Vout *para una medida sin trigger de 1 semiciclo en el puerto serie* COM3.

Para la siguiente figura número 70, se han representado 5 semiciclos tanto de corriente de entrada como de voltajes de entrada y salida. Aquí se puede apreciar que, debido a la auto escalabilidad del eje X, ahora la forma de los distintos puntos se asemeja más a una línea.

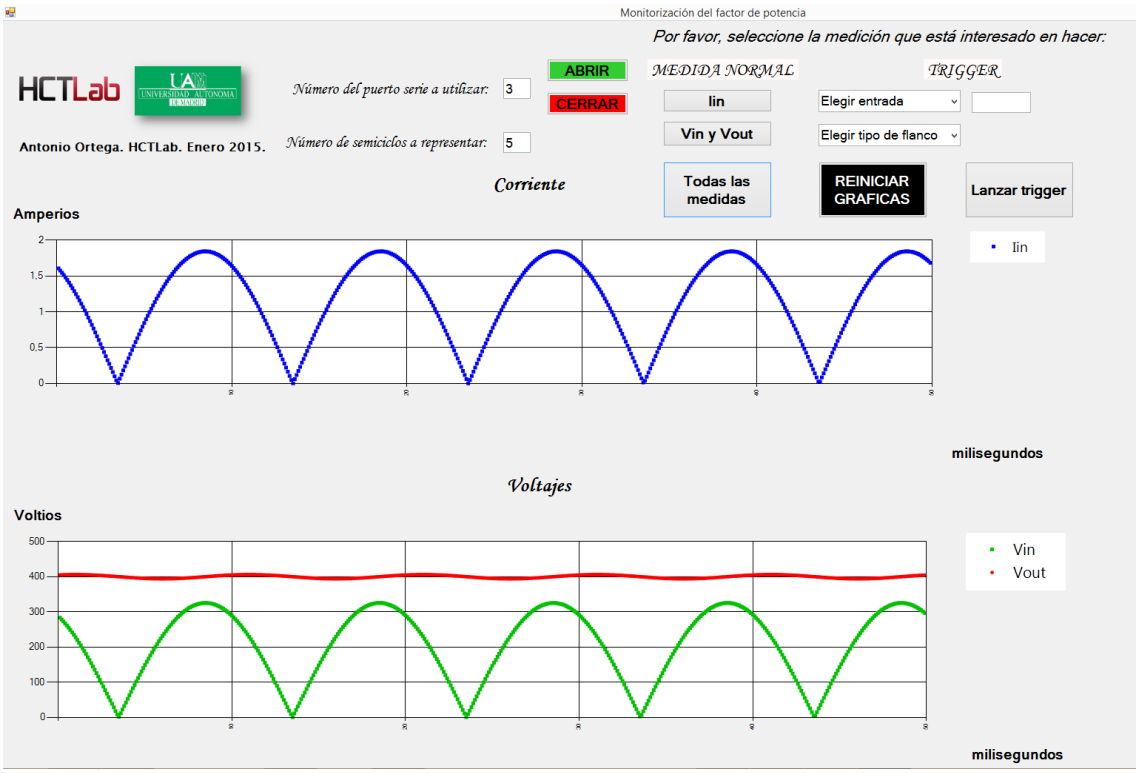

Fig. 70: Gráficas de Iin, Vin *y* Vout *para una medida sin trigger de 5 semiciclos en el puerto serie* COM3.

Se comentaba en el capítulo 6 que el usuario podía elegir representar sólo una de las gráficas, ya que por ejemplo podría estar interesado en observar sólo los valores de los voltajes. Esto es lo que se aprecia en la figura 71, donde sólo se representan 4 semiciclos de Vin y Vout.

Fig. 71: Gráficas de Vin *y* Vout *para una medida sin trigger de 4 semiciclos en el puerto serie* COM3.

7.2. Resultados de medidas con *trigger*

Si se desea implementar la funcionalidad de *trigger*, se ha de seleccionar en primer lugar los valores deseados en el menú *trigger* como se comentaba en el capítulo 6. Al igual que para el caso de las capturas sin *trigger*, el puerto serie utilizado ha sido el *COM3*.

En la figura número 72, se ha realizado una captura de 1 semiciclo de Iin, Vin y Vout, para un trigger de Iin = 0,3 amperios, con flanco positivo. Como se puede observar, estos valores de *trigger* introducidos son mostrados en la parte superior derecha de la interfaz, así el usuario tiene en todo momento confirmación de la medida que está llevando a cabo.

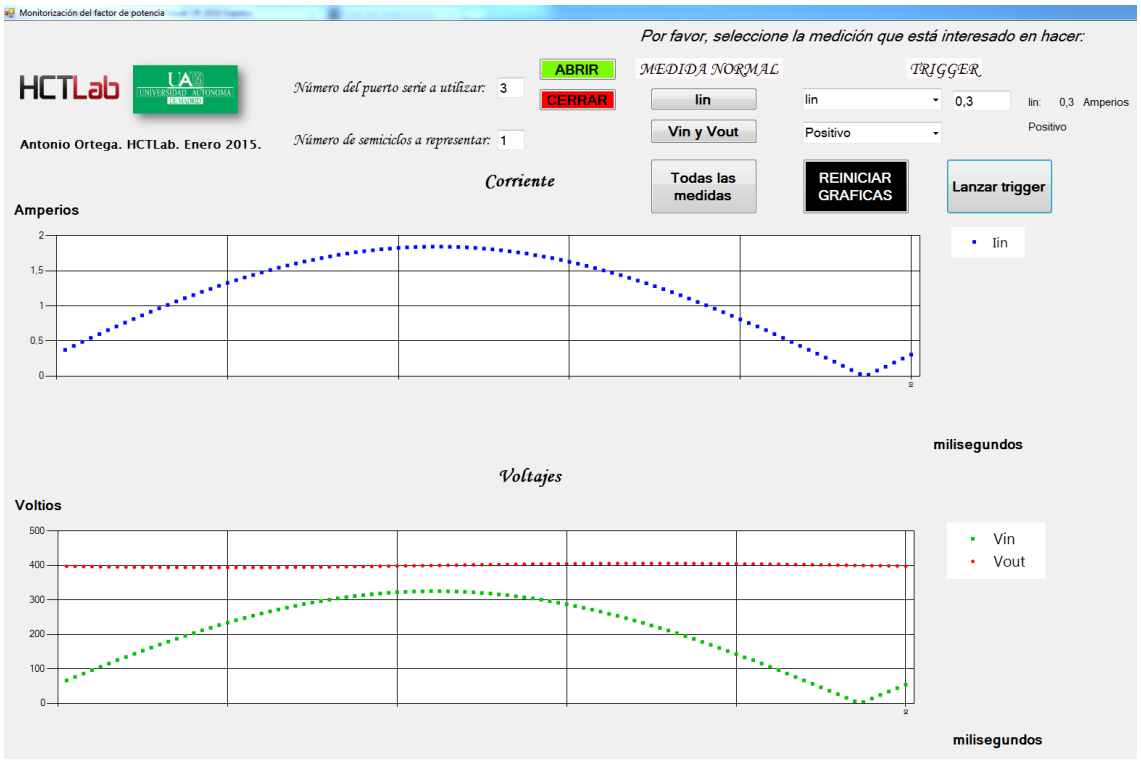

Fig. 72: Gráficas de Iin, Vin *y* Vout *para una medida con trigger de flanco positivo de valor Iin= 0,3 A y de 1 semiciclo en el puerto serie* COM3.

En las siguientes figuras número 73 y 74, se muestran sendas capturas para dos *triggers* distintos: por un lado un trigger con flanco negativo del voltaje de entrada con valor 200 voltios, y por otro lado un trigger con flanco positivo de la corriente de entrada con valor 1 amperios. En ambos casos se han representado 7 semiciclos de red.

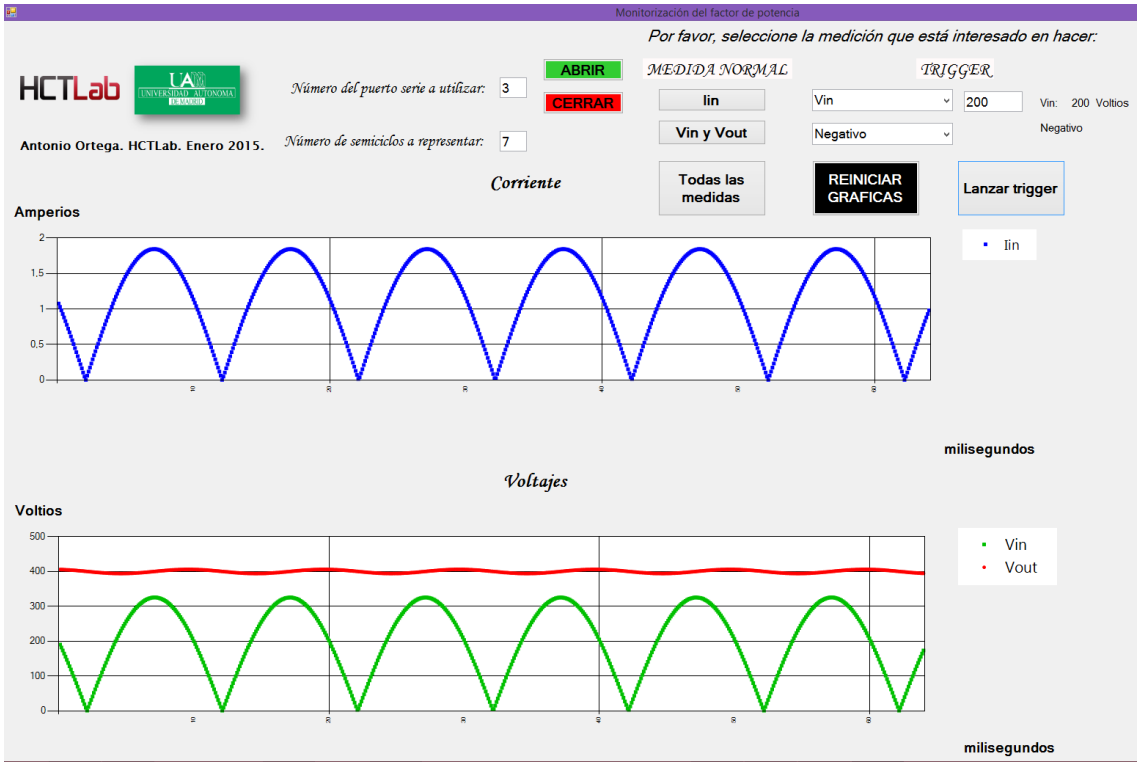

Fig. 73: Gráficas de Iin, Vin *y* Vout *para una medida con trigger de flanco negativo de valor Vin= 200 V y de 7 semiciclos en el puerto serie* COM3.

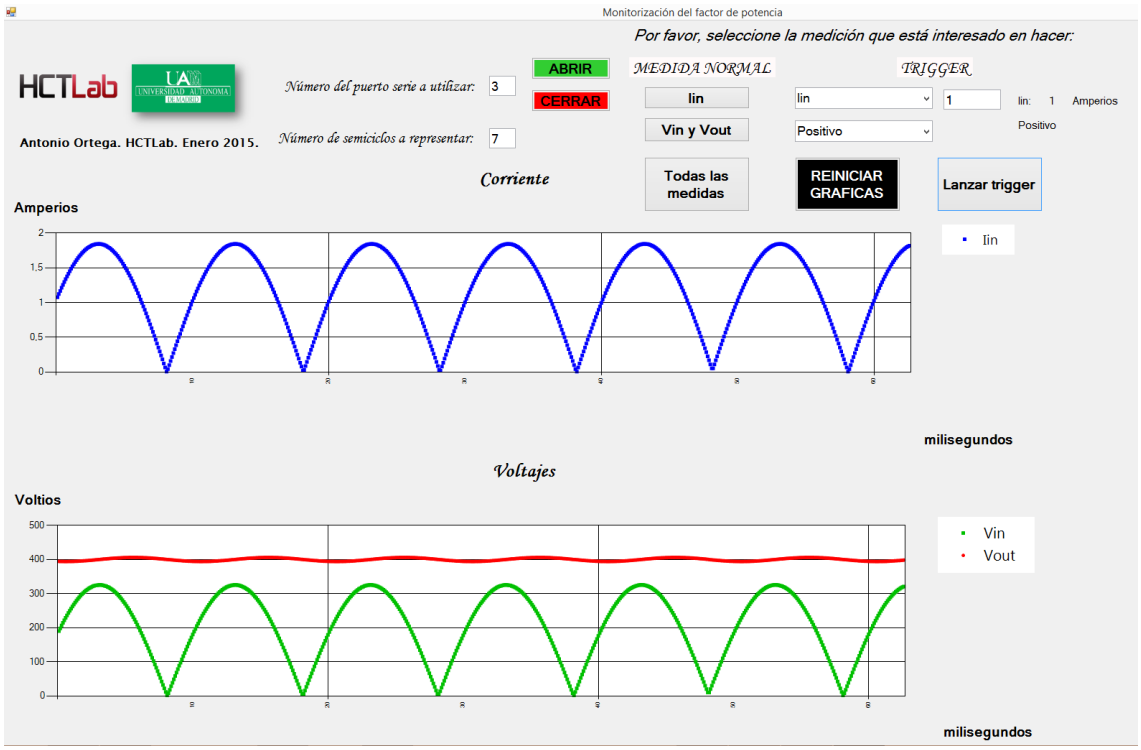

Fig. 74: Gráficas de Iin, Vin *y* Vout *para una medida con trigger de flanco positivo de valor Iin= 1 A y de 7 semiciclos en el puerto serie* COM3.

Para comprobar el correcto funcionamiento del *trigger*, se llevó a cabo la siguiente mejora en el código VHDL: a pesar que los valores de las memorias precalculadas que se utilizan para mandar los datos desde la FPGA oscilan entre 0 y 2 amperios aproximadamente para el caso de la corriente de entrada, entre 0 y 350 voltios aproximadamente para el caso del voltaje de entrada, y entre 380 y 420 voltios para el caso del voltaje de salida, se implementó una funcionalidad que hiciese que cuando el usuario pulse un botón en la FPGA, estos valores se vean incrementados. Concretamente, Iin se ve incrementada en +1 Amperio, y Vin y Vout se ven incrementados ambos en +50 voltios. Dicho código VHDL se puede observar en la figura 75. El botón de la Spartan 3 utilizado como `botonTrigger` ha sido el número *M13*.

```
tensionEntradaAux <=  (tensionEntrada + 50*8) when botonTrigger = '1'   else    tensionEntrada;
tensionSalidaAux <=  (tensionSalida  + 50*4) when botonTrigger = '1'    else tensionSalida;
CorrienteEntradaAux <= (CorrienteEntrada  + 512) when botonTrigger = '1'    else    CorrienteEntrada  ;
```

Fig. 75: Código en VHDL para incrementar los valores de Iin, Vin y Vout.

Si se fija un *trigger* con flanco positivo de Iin = 2 A, nunca se representaría ningún dato a no ser que se pulse el botón anteriormente programado y se obtengan valores de Iin superiores a 2 amperios. De esta manera se comprueba el correcto funcionamiento del *trigger*. Esto es lo que se observa en la figura 76.

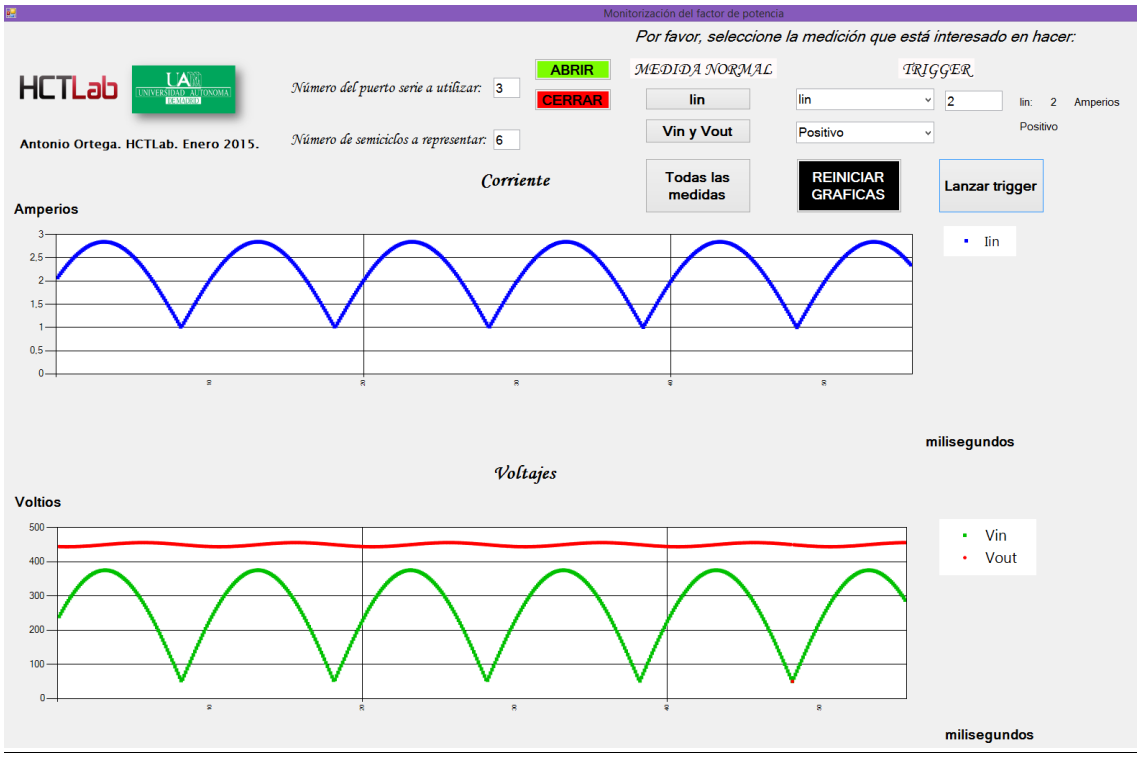

Fig. 76: Gráficas de Iin, Vin *y* Vout *para una medida con trigger de flanco positivo de valor Iin= 2 A y de 6 semiciclos en el puerto serie* COM3.

8. CONCLUSIONES

Los convertidores de potencia conmutados presentan grandes ventajas debido a que mejoran de manera significativa le eficiencia en la conversión de corrientes y tensiones. A su vez, estos convertidores de potencia conmutados necesitan de un sistema de control el cual puede ser analógico o digital. El control digital es el más extendido hoy día debido a sus ventajas con respecto al control analógico en la reprogramación de los dispositivos, la fiabilidad y la integración de los mismos.

Otra de las grandes ventajas del control digital, es la facilidad con la que se pueden añadir interfaces de comunicación entre el control y un dispositivo externo, para la monitorización y configuración del convertidor.

En particular, este TFG ha mostrado el diseño de un sistema de monitorización para un convertidor elevador actuando como corrector de factor de potencia. Este sistema se ha segmentado en distintos módulos que luego se han interconectado entre sí.

En primer lugar se ha mostrado el *módulo de comunicación*, el cual transporta la información digitalizada del convertidor de potencia y la lleva hacia el módulo UART-USB. El dispositivo fundamental de este módulo es la FPGA Spartan 3 S*tarter Board*, dentro de la cual se ha llevado a cabo la construcción de la trama de 7 bytes con los datos de corrientes y voltajes originados en el convertidor de potencia. Esta FPGA manda constantemente las tramas por semiciclo de red construidas mediante VHDL a la interfaz UART-USB a una velocidad de 3 Mbaudios, gracias a sus módulos VHDL que implementan la lógica reprogramable de una UART. Debido a que un convertidor de potencia maneja cientos de vatios, el estar monitorizando constantemente dicho convertidor podría suponer un riesgo a las personas, es por ello que el prototipado del sistema (la trama de 7 bytes construida), se elabora a partir de unas memorias precalculadas en VHDL basándose en la principio de periodicidad de las señales de corriente y voltajes de un convertidor de potencia. No obstante, el sistema no necesita ningún cambio para trasladarlo a un convertidor de potencia real.

En segundo lugar se explicado el *módulo UART-USB*, el cual está compuesto por un PCB donde se integra el dispositivo FT232R. Con este dispositivo se transportan las tramas de 7 bytes provenientes de los módulos VHDL de la FPGA que implementaban la lógica de una UART hacia un conector USB que se conecta al puerto serie del ordenador.

Las pruebas experimentales que se han realizado en el *módulo de la interfaz gráfica* con la aplicación programada en C#, demuestran finalmente la viabilidad del diseño, por un bajo coste en el sistema (aproximadamente unos 7,35 € con todos los componentes del PCB).

BIBLIOGRAFÍA

[1] Ned Mohan, "*Power electronics: a first course*". Editorial: John Wiley & Sons, Inc. ISBN: 978-1-118-07480-0

[2] Linear Technology, "*Power conversion from Miliamps to Amps at Ultra-High Efficiency*", http://cds.linear.com/docs/en/application-note/an54af.pdf Última consulta: 16/01/2015.

[3] Ángel de Castro Martín, "*Aplicación del control digital basado en hardware específico para convertidores de potencia conmutados*", Tesis Doctoral, Universidad Politécnica de Madrid, 2003.

[4] Alberto Sánchez González, "Aportaciones mediante implementación basada en sistemas embebidos al control digital de convertidores conmutados". Tesis doctoral, Universidad Autónoma de Madrid, 2013

[5] "*Factor de potencia*", Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Factor_de_potencia. Última consulta: 16/01/2015.

[6] Víctor Manuel García López, "Diseño y creación de un convertidor elevador con las etapas de potencia, actuación y sensado". Trabajo fin de grado, Universidad Autónoma de Madrid, 2014.

[7] Robert W. Erickson, Dragan Maksimovic, "*Fundamentals of power electronics*". Editorial: Springer Sciencie+Business Media LLC. 2ª edición, año 2002. ISBN: 978-1475705591

[8] PicoBlaze 8-bit controller, http://www.xilinx.com/products/intellectual-property/picoblaze.html, Última fecha de consulta: 10/01/2015.

[9] M.J.M. Pelgrom, "*Analog-to-Digital Conversion*". Editorial: Springer New York. Edición: 2013. ISBN: 978-1-4614-1370-7.

[10] Material de D.José Colás, asignatura "*Sistemas Electrónico de Dispositivos*".

[11] "*UART*". Wikipedia, la enciclopedia libre. http://es.wikipedia.org/wiki/Universal_Asynchronous_Receiver-Transmitter. Última consulta: 14/01/2015.

[12] Ken Chapman, "*UART transmitter and receiver macros*". Xilinx Ltd. 2003.

[13] Xilinx, "*Spartan 3 FPGA Family Data Sheet*". http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf. Junio 2007. Última fecha de consulta: 17/01/2015.

[14] Farnell, tienda online de componentes electrónicos: http://es.farnell.com. Última fecha de visita: 14/01/2015.

[15] Joseph Albahari y Ben Albahari, "*C# 5.0 in a Nutshell*". Editorial: O'Reilly Media, Inc. Junio 2012, 5ª edición. ISBN: 978-1-449-32010-2.

[16] "C Sharp", Wikipedia, la enciclopedia libre: http://es.wikipedia.org/wiki/C_Sharp. Última fecha de visita: 14/01/2015.

[17] "*C# Programming Guide*", Microsoft: http://msdn.microsoft.com/es-es/library/67ef8sbd.aspx. Última fecha de visita: 14/01/2015.

[18] "*Microsoft MSDN Library*", http://msdn.microsoft.com/en-us/library/ms123401.aspx, Microsoft. Última fecha de consulta: 18/01/2015.

ANEXO I: GLOSARIO

AC	*Alternating Current*
ADC	*Analog to Digital Converter*
BC	*Boost Converter*
BL	*Boltom Layer*
DC	*Direct Current*
DCM	*Digital Clock Management*
EEPROM	*Electrically Erasable Programmable Read Only Memory*
FPGA	*Field-Programmable Gate Array*
FSM	*Finite State Machine*
FTDI	*Future Technology Devices International*
GUI	*Graphical User Interface*
I/O	*Input/Output*
JTAG	*Joint Test Action Group*
LSB	*Less Significant Byte*
LUT	*Look-Up Table*
MCU	*Microcontroller*
MSB	*Most significant Byte*
PCB	*Printed Circuit Board*
QFN	*Quad Flat No-Lead*
RAM	*Random Access Memory*
SMD	*Surface-Mount Device*
SSOP	*Shrink Small Outline Package*
TFG	*Trabajo Fin de Grado*
TL	*Top Layer*
UART	*Universal Asynchronous Receiver/Transmitter*

ANEXO II: LISTA DE CÓDIGOS

En este anexo se procede a presentar el código implementado en la elaboración de este Trabajo Fin de Grado. Se divide principalmente en 2 secciones: la parte de programación en VHDL, la cual se lleva a cabo en Xilinx ISE, y se programa en la FPGA; y la parte de programación en C#, la cual se lleva a cabo en el entorno de desarrollo *Microsoft Visual Studio* y se simula en el mismo.

Los ficheros incluidos son:

- Proyecto en VHDL:

- *Top_level.vhd*: *Top level* del diseño en VHDL donde tiene lugar la construcción de la trama de 7 bytes, su envío a través de los módulos reprogramables que implementan la funcionalidad de la UART, así como la gestión de las memorias precalculadas con los valores de Vin, Vout e Iin.
- *Top_level.ucf:* para la gestión de los pines entrada y salida de la FPGA Spartan 3.
- *testBench_UART*: *Test Bench* en VHDL simple para poder llevar a cabo una simulación básica.

- Proyecto en C#:

- *Form1.cs*: Donde se lleva a cabo la gestión y graficación de las tramas de 7 bytes recibidas por el puerto serie.

II.1. Proyecto en VHDL

II.1.1) Top_level.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

library UNISIM;
use UNISIM.VComponents.all;

entity TOP_Level_v1 is
    port (
                  -- Reloj activo flanco subida
                Clk50MHz      : in  std_logic;
                  -- Reset asincrono activo nivel alto
                Reset         : in  std_logic;
                  -- Dato de entrada desde el PC

                Serial_in     : in  std_logic;
                  -- Dato de salida hacia el PC
                Serial_out    : out std_logic;
                  -- Para encender un led de prueba en caso de que
            recibamos el dato correcto.
                Led           : out std_logic;
```

```
                Clk_uart_debug :  out std_logic ;
                LedErrorBufferFull : out std_logic;
                write_buffer_debug :  out std_logic ;
                botonTrigger: in std_logic

    );

end TOP_Level_v1;

architecture Behavioral of TOP_Level_v1 is

-----------------------------------------------
---- DECLARACIONES DE LOS MODULOS UART ------
-----------------------------------------------

COMPONENT uart_tx is
    port (
                      data_in : in std_logic_vector(7 downto 0);
                write_buffer : in std_logic;
                reset_buffer : in std_logic;
                en_16_x_baud : in std_logic;
                  serial_out : out std_logic;
                 buffer_full : out std_logic;
            buffer_half_full : out std_logic;
                         clk : in std_logic

    );
 end COMPONENT;


 COMPONENT uart_rx is
    port (
                       serial_in : in std_logic;
                        data_out : out std_logic_vector(7 downto 0);
                    read_buffer : in std_logic;
                   reset_buffer : in std_logic;
                   en_16_x_baud : in std_logic;
             buffer_data_present : out std_logic;
                    buffer_full : out std_logic;
               buffer_half_full : out std_logic;
                           clk : in std_logic

    );
 end COMPONENT;

-----------------------------------------------
---------- DECLARACIONES DEL DCM ------------
-----------------------------------------------

    COMPONENT mult_reloj
    PORT(
        CLKIN_IN : IN std_logic;
        RST_IN : IN std_logic;
        CLKFX_OUT : OUT std_logic;
        CLKIN_IBUFG_OUT : OUT std_logic;
        CLK0_OUT : OUT std_logic;
        CLK2X_OUT : OUT std_logic;
        LOCKED_OUT : OUT std_logic
        );
    END COMPONENT;
```

```
------------------------------------------------------------------
---------- DECLARACIONES DE LAS MEMORIAS PRECALCULADAS ------------
------------------------------------------------------------------

COMPONENT seno_mem is --Seno para  V_in
      port(
      signal DO         :out std_logic_vector(15 downto 0);       -- 16-
bit Data Output
      signal ADDR       :in std_logic_vector(9 downto 0);         -- 10-
bit Address Input
      signal CLK        :in std_logic;                             --
Clock
      signal EN         :in std_logic;                             --
RAM Enable Input
      signal SSR        :in std_logic;                             --
Synchronous Set/Reset Input
      signal WE         :in std_logic                              --
Write Enable Input
      );
end COMPONENT;

COMPONENT seno_mem_Iin is --Seno para I_in
      port(
      signal DO         :out std_logic_vector(15 downto 0);       -- 16-
bit Data Output
      signal ADDR       :in std_logic_vector(9 downto 0);         -- 10-
bit Address Input
      signal CLK        :in std_logic;                             --
Clock
      signal EN         :in std_logic;                             --
RAM Enable Input
      signal SSR        :in std_logic;                             --
Synchronous Set/Reset Input
      signal WE         :in std_logic                              --
Write Enable Input
      );
end COMPONENT;


COMPONENT seno_mem_Vout is --Seno para V_out
      port(
      signal DO   : out std_logic_vector(15 downto 0);        -- 16-bit
Data Output
      signal ADDR :in std_logic_vector(9 downto 0);         -- 10-bit
Address Input
      signal CLK  :in std_logic;                             -- Clock
      signal EN   :in std_logic;                                -- RAM
Enable Input
      signal SSR  :in std_logic;                            -- Synchronous
Set/Reset Input
      signal WE   :in std_logic                                 -- Write
Enable Input
      );

end COMPONENT;


-------------------------------------------
--------- DECLARACION DE SEÑALES ----------
-------------------------------------------
```

```
    signal botonTriggerSignal : std_logic;
    signal SoloUnaVez: integer := 0;

    signal Clk: std_logic;
    signal enable16x_uart : std_logic;
    signal contador16x_uart: std_logic_vector(8 DOWNTO 0):=(others =>
'0');
    signal    contador_Write_buffer_tx:    std_logic_vector(16    DOWNTO
0):=(others => '0');

    signal clk_UART, clk_UARTR : std_logic;


    --- señales para TX ---
    signal write_buffer_tx: std_logic;
    signal reset_buffer_tx: std_logic;

    signal DATA_In_uart_tx:  std_logic_vector(7 downto 0);
    signal buffer_full_tx: std_logic;
    signal buffer_half_full_tx: std_logic;

    --- Señales para RX ---
    signal read_buffer_rx: std_logic;
    signal reset_buffer_rx: std_logic;
    signal buffer_full_rx: std_logic;
    signal buffer_half_full_rx: std_logic;
    signal buffer_data_present_rx: std_logic;
    signal data_out_rx: std_logic_vector(7 downto 0);


    constant   ITERACIONES_uart:   std_logic_vector(8   downto   0)   :=
conv_std_logic_vector(7, 9);
    constant ITERACIONES_write_buffer: std_logic_vector(16 downto 0) :=
conv_std_logic_vector(65200, 17);

-----------------------------------------------------------------------
--- Señales que vamos a usar para la composición y el envío de las tramas
-----------------------------------------------------------------------

    signal tensionEntrada: std_logic_vector(15 downto 0); --El ADC nos
da unos 8-10 bits en datos. Ponemos 16 bits para estar seguros.
    signal tensionSalida: std_logic_vector(15 downto 0);
    signal CorrienteEntrada: std_logic_vector(15 downto 0);

    signal tensionEntradaAux: std_logic_vector(15 downto 0); -- Señales
auxiliares para el caso de que se necesiten aumentar los valores de
tensiones o corrientes pulsando el botón de la FPGA.
    signal tensionSalidaAux: std_logic_vector(15 downto 0);
    signal CorrienteEntradaAux: std_logic_vector(15 downto 0);
    signal  contadorUart_tx_frame:  std_logic_vector(26  downto  0);  --
Contador que vamos a usar para llegar a los 10ms para enviar la trama.
    constant  ITERACIONES_MAX_TRAMA:  integer  :=  10000;  --  Tope  del
contador que se me mencionaba en la línea de arriba.
    constant  ITERACIONES_MAX_MEMORIA:  integer  :=  1000;   --  Tope  del
contador para la generación de las memorias con los semiciclos.

    signal contadorTipoDato: std_logic_vector(1 downto 0); -- Este es
el contador que vamos a usar para introducir los 3 tipos de datos
distintos V_in, V_out y I_in
```

```
    signal uart_tx_frame_ready: std_logic; -- Flag que indicará cuando
hemos introducido los 7 bytes de la trama (6 Datos  + 1 checksum) y que
esta lista para enviar.

    -- Máquina de estados (FSM) en la que vamos a usar para estar
esperando la trama (IDLE) y para enviar los 7 bytes de la trama mediante
7 estados distintos.
    type  STATE_TYPE  is  (IDLE,  SEND_Vin1,  SEND_Vin2,  SEND_Vout1,
SEND_Vout2, SEND_Iin1, SEND_Iin2, SEND_Checksum);
   signal Estado   : STATE_TYPE;
    signal  contadorMaquinaEstados:  std_logic_vector(3  downto  0);  --
Contador que usaremos para ir de 0 a 6, para enviar los 7 bytes de la
trama por UART_TX

    type UART_ARRAY is  array (6 downto 0) of std_logic_vector(7 downto
0); -- Array de tramas a construir de 1 byte cada una.
    signal uart_tx_frame: UART_ARRAY;
    constant  ITERACIONES_MAX_ARRAY:  std_logic_vector(2  downto  0)  :=
conv_std_logic_vector(4, 3);

    ------------------------
    ------ SEMICICLO ------
    ------------------------
    signal cntMemoriaAddr : integer range 0 to 1000;
    signal addrMem_Vin: std_logic_vector(9 downto 0);
    signal addrMem_Vout: std_logic_vector(9 downto 0);
    signal addrMem_Iin: std_logic_vector(9 downto 0);

    constant ITERACIONES_MAX_SENO: integer := 1000;
begin
 ---- ----  ---- ----  INSTANCIACIONES ---- ----  ---- ----

 -- Instanciación del DCM utilizado para el convertidor de potencia.
    Inst_mult_reloj: mult_reloj PORT MAP(
        CLKIN_IN => Clk50MHz,
        RST_IN => reset,
        CLKIN_IBUFG_OUT => open,
        CLKFX_OUT => clk_UART,
        CLK0_OUT => open,
        CLK2X_OUT => Clk, -- Clk a 100Mhz
        LOCKED_OUT => open
    );

 -- Instanciaciones de las memorias precalculatas para Vin, Vout e Iin.
    INSTANCIACION_seno_Vin: seno_mem
    port map(
        DO      => tensionEntrada,
        ADDR     => addrMem_Vin, --Hay que sumarle + 1, cada 10 us (ya
que el reloj que tenemos original es de 100Mhz)
        CLK => Clk,
        EN      => '1',
        SSR => Reset,
        WE      => '0'
        );

    INSTANCIACION_seno_Vout: seno_mem_Vout
    port map(
        DO      => tensionSalida,
        ADDR     => addrMem_Vin, --Hay que sumarle + 1, cada 10 us (ya
que el reloj que tenemos original es de 100Mhz)
        CLK => Clk,
```

```
        EN      => '1',
        SSR => Reset,
        WE      => '0'
        );

    INSTANCIACION_seno_Iin: seno_mem_Iin
    port map(
        DO      => CorrienteEntrada,
        ADDR     => addrMem_Vin, --Hay que sumarle + 1, cada 10 us (ya
que el reloj que tenemos original es de 100Mhz)
        CLK => Clk,
        EN      => '1',
        SSR => Reset,
        WE      => '0'
        );

-- Instanciaciones de los módulos VHDL que implementan la lógica
reprogramable de una UART.
 INSTANCIACION_uart_tx: uart_tx
 port map (
                   data_in => DATA_In_uart_tx,
                 write_buffer => write_buffer_tx,--write_buffer_tx, --
señal ACTIVO BAJO
                 reset_buffer => Reset,
                 en_16_x_baud => enable16x_uart,
                   serial_out => Serial_out,
                  buffer_full => buffer_full_tx,
              buffer_half_full => buffer_half_full_tx,
                          clk => Clk

 );

  INSTANCIACION_uart_rx: uart_rx
 port map (
                             serial_in => Serial_in ,
                      data_out => data_out_rx,
                    read_buffer => read_buffer_rx,
                   reset_buffer => Reset,
                   en_16_x_baud => enable16x_uart,
             buffer_data_present => buffer_data_present_rx,
                    buffer_full => buffer_full_rx,
                buffer_half_full => buffer_half_full_rx,
                            clk => Clk

 );
 ---- ----  ---- ----  ---- ----  ---- ----  ---- ----

-- Proceso con el que se enciende un LED en caso de que el buffer de
transmisión se llene.
--  Este hipotetico caso no deberia cumplirse si todo funciona
correctamente.
 process(Clk,Reset)
 begin
    if Reset = '1' then
        LedErrorBufferFull <= '0';
    elsif rising_edge(Clk) then
        if buffer_full_tx = '1' then
            LedErrorBufferFull <= '1';
        end if;
    end if;
end process;
```

```
-- Proceso para el contador 16x del reloj de la UART

 process(Clk, Reset)
 begin
    if Reset = '1' then
        enable16x_uart <= '0';
        clk_UARTR <= '0';
    elsif rising_edge(Clk) then
        clk_UARTR <= clk_UART;
        if clk_UART = '1' and clk_UARTR = '0' then
            enable16x_uart <= '1';
        else
            enable16x_uart <= '0';
        end if;
    end if;
 end process;


-- Proceso de encendido del LED y gestiona lo que entra desde el PC
-- en caso de que este mande información de vuelta
process (Clk, Reset)
begin
   if Reset = '1' then
            LED <= '0';
            read_buffer_rx <= '0';

    elsif rising_edge(Clk) then
        if buffer_data_present_rx = '1' then
            read_buffer_rx <= '1'; -- asi doy el dato por leido.
            if data_out_rx = x"42" then -- Si recibo una 'B' en ascii
enciendo el Led.
                LED <= '1';
            elsif data_out_rx = x"43" then   -- Si recibo una 'C' en
ascii apago el Led.
                LED <= '0';
            end if;
        else
            read_buffer_rx <= '0'; -- Quiere decir que el dato aun no
lo he leido, por si viene otro buffer_data_present_rx.
        end if;
   end if;
end process;

Clk_uart_debug <= enable16x_uart;
write_buffer_debug <= write_buffer_tx;


---- Código para mandar tramas con los valores de Vin, Vour e In -----
--- Proceso en el que vamos a ir asignando valores  distintos de V_in,
V_ou y I_in a través de las memorias precalculadas.

process (Clk, Reset)
begin
    if Reset = '1' then
        cntMemoriaAddr <= 0;
        addrMem_Vin <= (others => '0');
        addrMem_Vout <= (others => '0');
        addrMem_Iin <= (others => '0');
    elsif rising_edge (Clk) then
```

```
        -- Si queremos tener una direccion addrMem cada 10us, y nuestro
Clk original es de 100Mhz, tenemos que hacer un contador hasta 1000
        if cntMemoriaAddr = ITERACIONES_MAX_MEMORIA then
            if addrMem_Vin = conv_std_logic_vector(1000,10) then
                addrMem_Vin <= (others => '0');
            else
                addrMem_Vin <= addrMem_Vin + 1;
            end if;
            cntMemoriaAddr <= 0;
        else
            cntMemoriaAddr <= cntMemoriaAddr + 1;
        end if;


    end if;
end process;

-- En caso de que se presione el botón M13 de la FPGA, los valores de
Iin se incrementara en +1 amperio, y para el caso de Vin y Vout en + 50
voltios.
tensionEntradaAux <=   (tensionEntrada + 50*8) when botonTrigger = '1'
else    tensionEntrada;
tensionSalidaAux <=   (tensionSalida   + 50*4) when botonTrigger = '1'
else tensionSalida;
CorrienteEntradaAux <= (CorrienteEntrada  + 512) when botonTrigger = '1'
else    CorrienteEntrada  ;


--- Proceso para rellenar la trama uart_tx_frame
process (Clk, Reset)
begin
    if Reset = '1' then
        for i in 0 to 6 loop
            uart_tx_frame(i) <= (others => '0');
        end loop;

        contadorTipoDato <= (others => '0');
        contadorUart_tx_frame <= (others => '0');
        uart_tx_frame_ready <= '0';

    elsif rising_edge(Clk) then
        if conv_integer(contadorUart_tx_frame) = ITERACIONES_MAX_TRAMA
then

                    contadorUart_tx_frame <= (others => '0');
                       -- La estructura va a ser:
                       -- 1er y 2° byte:  2 bytes de datos de V_in
                       -- 3° y 4° byte:  2 bytes de datos de V_out
                       -- 5° y 6° byte: 2 bytes de datos de I_in
                       -- 7° byte: CRC (suma de los bytes anteriores).
                    uart_tx_frame(0)  <= tensionEntradaAux(15 downto 8);
                    uart_tx_frame(1)  <= tensionEntradaAux(7 downto 0);
                    uart_tx_frame(2)  <= tensionSalidaAux(15 downto 8);
                    uart_tx_frame(3)  <= tensionSalidaAux(7 downto 0);
                    uart_tx_frame(4)  <= CorrienteEntradaAux(15 downto 8);
                    uart_tx_frame(5)  <= CorrienteEntradaAux(7 downto 0);
                    uart_tx_frame(6)  <= tensionEntradaAux(15 downto 8) +
tensionEntradaAux(7  downto  0)  +  tensionSalidaAux(15  downto  8)  +
tensionSalidaAux(7  downto  0)  +  CorrienteEntradaAux(15  downto  8)  +
CorrienteEntradaAux(7 downto 0);
```

```
                uart_tx_frame_ready <= '1'; -- Flag de activación para
indicar a la máquina de estados que tiene que pasar a enviar los 7 bytes
de la trama.
        else
            contadorUart_tx_frame <= contadorUart_tx_frame + 1;
            uart_tx_frame_ready <= '0'; --Solo cuando la trama entera
esté compuesta nos interesa avisar a la máquina de estados de que coja
el dato.
        end if;
    end if;
end process;

------------- Proceso de la máquina de estados -------------
-- Contamos con 8 estados:
-- IDLE: estamos esperando a que haya una trama lista para enviar.
Sabremos que la trama esta lista para enviar con el FLAG de
uart_tx_frame_ready.
-- SEND_Vin1: envío del 1er byte de datos de Vin
-- SEND_Vin2: envío del 2° byte de datos de Vin
-- SEND_Vout1: envío del 1er byte de datos de Vout
-- SEND_Vout2:  envío del 2° byte de datos de Vout
-- SEND_Iin1: envío del 1er byte de datos de Iin
-- SEND_Iin2: envío del 2° byte de datos de Iin
-- SEND_Checksum: envío del byte de Checksum.

process (Clk, Reset)
begin
    if Reset = '1' then
        Estado <= IDLE;
        write_buffer_tx <= '0'; --

    elsif rising_edge(Clk) then
        case Estado is
            when IDLE =>
            -- En este estado nos encontramos esperando a que el bit
uart_tx_frame_ready nos indique que hemos recibido datos para enviar.
                if uart_tx_frame_ready = '1' then
                    Estado <= SEND_Vin1;
                else
                    Estado <= IDLE;
                    DATA_In_uart_tx <= (others => '0');
                    write_buffer_tx <= '0';

                end if;

            when SEND_Vin1 =>
                    write_buffer_tx <= '1';
                    DATA_In_uart_tx <= uart_tx_frame(0);
                    Estado <= SEND_Vin2;
            when SEND_Vin2 =>
                    DATA_In_uart_tx <= uart_tx_frame(1);
                    Estado <= SEND_Vout1;
            when SEND_Vout1 =>
                    DATA_In_uart_tx <= uart_tx_frame(2);
                    Estado <=SEND_Vout2;
            when SEND_Vout2 =>
                    DATA_In_uart_tx <= uart_tx_frame(3);
                    Estado <= SEND_Iin1;

            when SEND_Iin1 =>
                    DATA_In_uart_tx <= uart_tx_frame(4);
```

```
                    Estado <= SEND_Iin2;
                when SEND_Iin2 =>
                    DATA_In_uart_tx <= uart_tx_frame(5);
                    Estado <= SEND_Checksum;
                when SEND_Checksum =>
                    DATA_In_uart_tx <= uart_tx_frame(6);

                    Estado <= IDLE;
            end case;
        end if;
    end process;
    end Behavioral;
```

II.1.2) Top_level.ucf

```
#Created by Constraints Editor (xc3s1000-ft256-4) - 2014/10/01
NET "Clk50MHz" TNM_NET = Clk50MHz;
TIMESPEC TS_Clk50MHz = PERIOD "Clk50MHz" 20 ns HIGH 50%;

NET "Clk" LOC="T9";


# Pines del 'Header' de la placa FTDI.
# PARTE SUPERIOR
# 2 -> SIN CONECTAR
# 4 -> DSR
# 6 -> RTS
# 8 -> CTS
# 10-> DCD

# PARTE INFERIOR
# 1 -> GND HEADER
# 3 -> SIN CONECTAR
# 5 -> TXD
# 7 -> RXD
# 9 -> DTR


NET "botonTrigger" LOC= "M13";
NET "Reset" LOC="L14";
NET "Serial_in" LOC = "N7";  # Conector A1, pin 5
NET "Serial_out" LOC = "T8";  # Conector A1, pin 7
NET "Led" LOC = "K12";       # conectores led, pin K12
NET "Clk_uart_debug" LOC = "R6";# Conector A1, pin 9
NET "write_buffer_debug" LOC = "T5";# Conector A1, pin 11
NET "LedErrorBufferFull" LOC="P11";
```

II.1.3) testBench_UART.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY testBench_UART IS
END testBench_UART;

ARCHITECTURE behavior OF testBench_UART IS
    -- Component Declaration for the Unit Under Test (UUT)
    COMPONENT TOP_Level_v1
    PORT(
         Clk50MHz : IN  std_logic;
         Reset : IN  std_logic;
         Serial_in : IN  std_logic;
         Serial_out : OUT  std_logic;
         Led : OUT  std_logic;
         Clk_uart_debug : OUT  std_logic
        );
    END COMPONENT;

   --Inputs
   signal Clk50MHz : std_logic := '0';
   signal Reset : std_logic := '0';
   signal Serial_in : std_logic := '0';

    --Outputs
   signal Serial_out : std_logic;
   signal Led : std_logic;
   signal Clk_uart_debug : std_logic;

   -- Clock period definitions
   constant Clk50MHz_period : time := 2 ns;
   constant Clk_uart_debug_period : time := 2 ns;
BEGIN
    -- Instantiate the Unit Under Test (UUT)
   uut: TOP_Level_v1 PORT MAP (
          Clk50MHz => Clk50MHz,
          Reset => Reset,
          Serial_in => Serial_in,
          Serial_out => Serial_out,
          Led => Led,
          Clk_uart_debug => open
        );
   -- Clock process definitions
   Clk50MHz_process :process
   begin
        Clk50MHz <= '0';
        wait for Clk50MHz_period/2;
        Clk50MHz <= '1';
        wait for Clk50MHz_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      Reset <= '1'; wait for 10ns; Reset <= '0';
      wait;
   end process;
END;
```

II.2. Proyecto en C#

II.2.1) Form1.cs

```
using System;
using System.IO;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Threading;

namespace proyecto_v1
{
    public partial class Form1 : Form
    {
        private void Form1_Load(object sender, EventArgs e)
        {
        }
        private void textBox1_TextChanged(object sender, EventArgs e)
        {
        }
        private void textBox2_TextChanged(object sender, EventArgs e)
        {
        }
        private void button1_intensidad_Click(object sender, EventArgs
e)
        {
            // Botón para representar sólo la corriente de entrada.
            VariablesGlobales.botonPulsado = 1;
            VariablesGlobales.MedirCorriente = 1;
            LecturaDatos();
        }
        private void button3_todos_Click(object sender, EventArgs e)
        {
            // Botón para representar todos los valores.
            VariablesGlobales.botonPulsado = 1;
            VariablesGlobales.MedirTodo = 1;
            LecturaDatos();

        }
        private void button2_voltajes_Click(object sender, EventArgs
e)
        {
            // Botón para representar sólo los voltajes.
            VariablesGlobales.botonPulsado = 1;
            VariablesGlobales.MedirVoltajes = 1;
            LecturaDatos();
        }
        private void button4_reiniciar_Click(object sender, EventArgs
e)
        {
            // Boton que reinicia tanto las gráficas como los datos
almacenados en memoria.
            chart3.Series[0].Points.Clear();
            chart4.Series[0].Points.Clear();
```

```
            chart4.Series[1].Points.Clear();

            VariablesGlobales.V_in.Clear();
            VariablesGlobales.I_in.Clear();
            VariablesGlobales.V_out.Clear();
            VariablesGlobales.ListaBytesLeido.Clear();
            Trigger.Lista_Trigger_I_in.Clear();
            Trigger.Lista_Trigger_V_in.Clear();
            Trigger.Lista_Trigger_V_out.Clear();


            Trigger.BotonTRIGGER = 0;
            Trigger.ValorTrigger = 0;
            Trigger.Trigger_Iin = 0;
            Trigger.Trigger_Vin = 0;
            Trigger.Trigger_Vout = 0;
            Trigger.Flanco_positivo = 0;
            Trigger.Flanco_negativo = 0;
            Trigger.ContadorTramaValidaTrigger = 0;

            VariablesGlobales.MedirVoltajes = 0;
            VariablesGlobales.MedirCorriente = 0;
            VariablesGlobales.MedirTodo = 0;
            VariablesGlobales.BytesLeido = 0;

            // Limpieza de los datos que puedan permanecer en el
buffer de entrada del puerto serie.
            serialPort1.DiscardInBuffer();

        }
        private void chart3_Click(object sender, EventArgs e)
//Grafico de la corriente
        {
        }
        private void chart4_Click(object sender, EventArgs e)
//Grafico de los voltajes
        {
        }
        private void ReiniciarTrigger()
        {
            // En caso de estar en una situacion de trigger, si no se
encuentra el valor buscado, se reinician los datos almacenados,
            // se leen del puerto serie datos nuevos, y se sigue
buscando.
            VariablesGlobales.V_in.Clear();
            VariablesGlobales.I_in.Clear();
            VariablesGlobales.V_out.Clear();
            VariablesGlobales.ListaBytesLeido.Clear();
            Trigger.Lista_Trigger_I_in.Clear();
            Trigger.Lista_Trigger_V_in.Clear();
            Trigger.Lista_Trigger_V_out.Clear();

            Trigger.ContadorTramaValidaTrigger = 0;
            VariablesGlobales.bytesPedidos = 0;

        }
        private void LecturaSerialPort()
        {
            while (serialPort1.IsOpen == false) // Por si acaso el
puerto serie se hubiese cerrado por algún subproceso de Microsoft.
                serialPort1.Open();
```

```
            int bytesLeidos = 0;
            // Vamos a representar 100 puntos por seno, por 7 (numero
de bytes), por 5 (por añadir un margen de seguridad de lectura).
            VariablesGlobales.bytesPedidos =
VariablesGlobales.NumeroSenosArepresentar * 100 * 7 * 5;

            while (bytesLeidos < VariablesGlobales.bytesPedidos)
            {
                while (serialPort1.BytesToRead != 0 && bytesLeidos <
VariablesGlobales.bytesPedidos)
                {
                    // Bucle que se ejecuta mientras haya bytes en el
puerto serie para leer.
                    VariablesGlobales.BytesLeido =
serialPort1.ReadByte();

VariablesGlobales.ListaBytesLeido.Add(VariablesGlobales.BytesLeido );
                    bytesLeidos++;
                }
            }
            int TramaValida = 0;

            //Caso de trigger.
            if (Trigger.BotonTRIGGER == 1)
            {

                while (TramaValida == 0 &&
Trigger.RepresentarSoloUnaVez == 0)
                {
                    while (TramaValida == 0 &&
(Trigger.ContadorTramaValidaTrigger + 6 <
VariablesGlobales.ListaBytesLeido.Count))
                    {

                        Trigger.V_inByte1 =
VariablesGlobales.ListaBytesLeido.ElementAt(Trigger.ContadorTramaValid
aTrigger);
                        Trigger.V_inByte2 =
VariablesGlobales.ListaBytesLeido.ElementAt(Trigger.ContadorTramaValid
aTrigger + 1);
                        Trigger.V_outByte1 =
VariablesGlobales.ListaBytesLeido.ElementAt(Trigger.ContadorTramaValid
aTrigger + 2);
                        Trigger.V_outByte2 =
VariablesGlobales.ListaBytesLeido.ElementAt(Trigger.ContadorTramaValid
aTrigger + 3);
                        Trigger.I_inByte1 =
VariablesGlobales.ListaBytesLeido.ElementAt(Trigger.ContadorTramaValid
aTrigger + 4);
                        Trigger.I_inByte2 =
VariablesGlobales.ListaBytesLeido.ElementAt(Trigger.ContadorTramaValid
aTrigger + 5);
                        Trigger.CRC =
VariablesGlobales.ListaBytesLeido.ElementAt(Trigger.ContadorTramaValid
aTrigger + 6);

                        Trigger.suma_TODO = Trigger.V_inByte1 +
Trigger.V_inByte2 + Trigger.V_outByte1 + Trigger.V_outByte2 +
Trigger.I_inByte1 + Trigger.I_inByte2;
```

```
                        Trigger.suma_TODO_mod = Trigger.suma_TODO %
256;
                        if (Trigger.CRC == Trigger.suma_TODO_mod)
                        {
                            Trigger.ContadorTramaValidaTrigger =
Trigger.ContadorTramaValidaTrigger + 7;

                            Trigger.V_in = (Trigger.V_inByte1 * 256 +
Trigger.V_inByte2) / 8;
                            Trigger.V_out = (Trigger.V_outByte1 * 256
+ Trigger.V_outByte2) / 4;
                            Trigger.I_in = (Trigger.I_inByte1 * 256 +
Trigger.I_inByte2) / 512;


Trigger.Lista_Trigger_V_in.Add(Trigger.V_in);

Trigger.Lista_Trigger_V_out.Add(Trigger.V_out);

Trigger.Lista_Trigger_I_in.Add(Trigger.I_in);
                            int posicionDatoAnterior;
                            if (Trigger.Lista_Trigger_I_in.Count > 1)
                                posicionDatoAnterior =
Trigger.Lista_Trigger_I_in.Count - 2;
                            else
                                posicionDatoAnterior = 0;


                            if (Trigger.Trigger_Iin == 1)
                            {
                                if ((Trigger.Flanco_positivo == 1 &&
Trigger.I_in >= Trigger.ValorTrigger &&
Trigger.Lista_Trigger_I_in[posicionDatoAnterior] <
Trigger.ValorTrigger) || (Trigger.Flanco_negativo == 1 && Trigger.I_in
<= Trigger.ValorTrigger &&
Trigger.Lista_Trigger_I_in[posicionDatoAnterior] >
Trigger.ValorTrigger))
                                {
                                   // Trigger encontrado para Iin.
                                   TramaValida = 1;

                                }
                            }
                            else if (Trigger.Trigger_Vin == 1)
                            {
                                if ((Trigger.Flanco_positivo == 1 &&
Trigger.V_in >= Trigger.ValorTrigger &&
Trigger.Lista_Trigger_V_in[posicionDatoAnterior] <
Trigger.ValorTrigger) || (Trigger.Flanco_negativo == 1 && Trigger.V_in
<= Trigger.ValorTrigger &&
Trigger.Lista_Trigger_V_in[posicionDatoAnterior] >
Trigger.ValorTrigger))
                                {
                                    // Trigger encontrado para Vin.
                                        TramaValida = 1;
                                }
                            }
                            else if (Trigger.Trigger_Vout == 1)
                            {
                                if ((Trigger.Flanco_positivo == 1 &&
Trigger.V_out >= Trigger.ValorTrigger &&
```

```
Trigger.Lista_Trigger_V_out[posicionDatoAnterior] <
Trigger.ValorTrigger) || (Trigger.Flanco_negativo == 1 &&
Trigger.V_out <= Trigger.ValorTrigger &&
Trigger.Lista_Trigger_V_out[posicionDatoAnterior] >
Trigger.ValorTrigger))
                                {
                                    // Trigger encontrado para Vout.
                                        TramaValida = 1;
                                }
                            }
                        }
                        else
                        {
                            Trigger.ContadorTramaValidaTrigger++;
                        }
                    }

                    // Esta sección de código se encuentra para evitar
entrar en la funcion de representar datos varias veces mientras
                    // no se encuentre el trigger correcto.
                    if (Trigger.RepresentarSoloUnaVez == 0 &&
TramaValida == 1)
                    {
                        Trigger.BotonTRIGGER = 0;
                        Trigger.RepresentarSoloUnaVez = 1;
                        LecturaDatos();


                    }
                    else if (Trigger.RepresentarSoloUnaVez == 0)
                    {
                        ReiniciarTrigger();
                        LecturaSerialPort();
                    }
                }
            }
        }

        // Función en la que se va a llevar a cabo el tratamiento de
los datos leidos y su representación en las gráficas.
        private void LecturaDatos()
        {

            int Indice_I_in = 0;
            int Indice_V_in = 0;
            int Indice_V_out = 0;

            // Para poner los valores de milisegundos en el eje X.
            int contadorLabel = 0;
            while (contadorLabel < 50)
            {
                string leyenda = Convert.ToString(contadorLabel * 10);

chart3.ChartAreas[0].AxisX.CustomLabels.Add(contadorLabel * 90,
contadorLabel * 110, leyenda);

chart4.ChartAreas[0].AxisX.CustomLabels.Add(contadorLabel * 90,
contadorLabel * 110, leyenda);

                contadorLabel++;
            }
```

```
            // Numero de semiciclos a representar. Valor introducido
por el usuario.
            string numeroSenos = textBox_senos.Text;
            try
            {
                VariablesGlobales.NumeroSenosArepresentar =
Convert.ToInt16(numeroSenos);
            }
            catch (Exception ex)
            {
                MessageBox.Show("No se ha especificado un número de
semiciclos. Intentelo de nuevo.");
                return;
            }

            if (VariablesGlobales.botonPulsado == 1 )
            {
                int numeroPuntos = 0;

                if (Trigger.RepresentarSoloUnaVez == 0)
                    LecturaSerialPort(); //Para evitar volvernos a
meter en la lectura del puerto serie si ya ha habido trigger.

                int contadorTramaValida =
Trigger.ContadorTramaValidaTrigger; //Contador para ir moviéndonos por
la trama.
                // Este contador empezará por 0 si no hay trigger.

                // En caso de tener menos puntos de los solicitados
por el usuario, nos metemos en la lectura del puerto serie para
rellenar los datos.
                if (Trigger.RepresentarSoloUnaVez == 1 &&
VariablesGlobales.ListaBytesLeido.Count -
Trigger.ContadorTramaValidaTrigger   <
VariablesGlobales.NumeroSenosArepresentar * 100)
                {
                    Trigger.BotonTRIGGER = 0;
                    LecturaSerialPort();
                }
                while (numeroPuntos <
VariablesGlobales.NumeroSenosArepresentar*100 )
                {

                    // Ahora vamos a tener la siguiente estructura de
la trama.
                    // | V_in | V_out | I_in | CRC | --> 7 bytes en
total.
                    // V_in -> 2 bytes.
                    // V_out -> 2 bytes.
                    // I_in -> 2 bytes.
                    // CRC -> 1 byte.


                    // Declaracion de variables que vamos a usar en
este contexto.
                    double V_inByte1; // Cada uno de los 7 bytes en
los que se va a ir leyendo la trama.
                    double V_inByte2;
                    double V_outByte1;
                    double V_outByte2;
```

```
                    double I_inByte1;
                    double I_inByte2;
                    double CRC;
                    double I_in, V_in, V_out;


                    int tramaValida = 0; //Flag de terminacion del
bucle.


                    while (tramaValida == 0 && (contadorTramaValida +
6 < VariablesGlobales.ListaBytesLeido.Count) )
                    {
                        V_inByte1 =
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida);
                        V_inByte2 =
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 1);

                        V_outByte1 =
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 2);
                        V_outByte2 =
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 3);

                        I_inByte1 =
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 4);
                        I_inByte2 =
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 5);

                        CRC =
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 6);

                        int sumaTODO =
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida) +
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 1) +
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 2) +
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 3) +
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 4) +
VariablesGlobales.ListaBytesLeido.ElementAt(contadorTramaValida + 5);
                        int sumaTODO_mod = sumaTODO % 256;
                        int TodoCeros = 0;

                        if (V_inByte1 == 0 && V_inByte2 == 0 &&
V_outByte1 == 0 && V_outByte2 == 0 && I_inByte1 == 0 && I_inByte2 ==
1)
                        {
                            TodoCeros = 1;
                        }
                        else
                        {
                            TodoCeros = 0;
                        }
                        if (CRC == sumaTODO_mod && TodoCeros == 0 )
                        {
                            contadorTramaValida = contadorTramaValida
+ 7; // Para avanzar a la siguiente trama.
                            tramaValida = 1;

                            V_in = (V_inByte1*256 + V_inByte2)/8;
                            V_out = (V_outByte1*256 + V_outByte2)/4;
                            I_in = (I_inByte1 * 256 + I_inByte2)/512;
```

```
                            // chart4.Series[0] --> V_in
                            // chart4.Series[1] --> V_out
                            // chart3.Series[0] --> I in

                            //Todas las medidas//
                            if (VariablesGlobales.MedirTodo == 1)
                            {
                                VariablesGlobales.I_in.Add(I_in);
                                Indice_I_in =
VariablesGlobales.I_in.Count;

chart3.Series[0].Points.AddXY(Indice_I_in, I_in); //Representación del
valor en el gráfico.

                                VariablesGlobales.V_in.Add(V_in);
                                Indice_V_in =
VariablesGlobales.V_in.Count;

chart4.Series[0].Points.AddXY(Indice_V_in, V_in);  //Representación
del valor en el gráfico.

                                VariablesGlobales.V_out.Add(V_out);
                                Indice_V_out =
VariablesGlobales.V_out.Count;

chart4.Series[1].Points.AddXY(Indice_V_out, V_out);  //Representación
del valor en el gráfico.
                            }
                            //Corriente de entrada//
                            else if (VariablesGlobales.MedirCorriente
== 1)
                            {

                                VariablesGlobales.I_in.Add(I_in);
                                Indice_I_in =
VariablesGlobales.I_in.Count;

chart3.Series[0].Points.AddXY(Indice_I_in, I_in);  //Representación
del valor en el gráfico.
                            }
                            //Voltajes//
                            else if (VariablesGlobales.MedirVoltajes
== 1)
                            {
                                VariablesGlobales.V_in.Add(V_in);
                                Indice_V_in =
VariablesGlobales.V_in.Count;

chart4.Series[0].Points.AddXY(Indice_V_in, V_in);  //Representación
del valor en el gráfico.

                                VariablesGlobales.V_out.Add(V_out);
                                Indice_V_out =
VariablesGlobales.V_out.Count;

chart4.Series[1].Points.AddXY(Indice_V_out, V_out);  //Representación
del valor en el gráfico.
                            }
```

```
                        }
                        else
                        { // En caso de que se pierdan tramas, esta
lista nos permite monitorizar qué número de trama se ha perdido.

TramaPerdida.IndiceTramas.Add(contadorTramaValida);
                            contadorTramaValida++;
                        }
                    }
                    numeroPuntos++;
                }
            }
        }

        public Form1()
        {
            CheckForIllegalCrossThreadCalls = false;

            InitializeComponent();
        }
        private void textBox3_TextChanged(object sender, EventArgs e)
        {

        }
        private void checkedListBox1_SelectedIndexChanged(object
sender, EventArgs e)
        {

        }
        private void textBox5_TextChanged(object sender, EventArgs e)
        {

        }
        // SECCION DEL BOTON DE TRIGGER //
        private void button1_Click(object sender, EventArgs e)
        {
            // Lectura del valor introducido.
            string numeroSinPuntos = textBox_trigger.Text.Replace('.',
',');
            try
            {
                Trigger.ValorTrigger =
Convert.ToDouble(numeroSinPuntos);
            }
            catch (Exception ex)
            {
                MessageBox.Show("No se ha especificado un número
correcto de tensión/corriente.");
                return;
            }
            VariablesGlobales.botonPulsado = 1;
            Trigger.BotonTRIGGER = 1;

            // Lectura de los valores introducidos en los menús
desplegables.
            //  También tiene lugar la representación de esta
información en las etiquetas.
            if (comboBox_tipo.Text == "Iin")
            {
                label6_trigger.Text = "Iin:";
                label9_trigger.Text = "Amperios";
```

```
                Trigger.Trigger_Iin = 1;
                VariablesGlobales.MedirTodo = 1;
            }
            else if (comboBox_tipo.Text == "Vin")
            {
                label6_trigger.Text = "Vin:";
                label9_trigger.Text = "Voltios";
                Trigger.Trigger_Vin = 1;
                VariablesGlobales.MedirTodo = 1;
            }
            else if (comboBox_tipo.Text == "Vout")
            {
                label6_trigger.Text = "Vout:";
                label9_trigger.Text = "Voltios";
                Trigger.Trigger_Vout = 1;
                VariablesGlobales.MedirTodo = 1;
            }
            if (comboBox_flanco.Text == "Negativo")
            {
                label7_trigger.Text = "Negativo";
                Trigger.Flanco_negativo = 1;
            }
            else if (comboBox_flanco.Text == "Positivo")
            {
                label7_trigger.Text = "Positivo";
                Trigger.Flanco_positivo = 1;
            }

            label8_trigger.Text = textBox_trigger.Text;

            // Lectura del número de semiciclos a representar.
            string numeroSenos = textBox_senos.Text;
            try
            {
                VariablesGlobales.NumeroSenosArepresentar =
Convert.ToInt16(numeroSenos);
            }
            catch (Exception ex)
            {
                MessageBox.Show("No se ha especificado un número de
semiciclos. Intentelo de nuevo.");
                return;
            }


            LecturaSerialPort();

        }

        private void textBox1_TextChanged_1(object sender, EventArgs
e)
        {
        }
        private void label6_Click(object sender, EventArgs e)
        {
        }
        private void label1_Click(object sender, EventArgs e)
        {
        }
        private void button1_Click_1(object sender, EventArgs e)
        {
```

```
            // Botón para abrir el puerto serie con el número del
mismo.
            string numeroIntroducido = textBox_serialPort.Text;
            string nombrePuertoSerie = "COM" + numeroIntroducido;

            serialPort1.PortName = nombrePuertoSerie;
            serialPort1.Open();
        }
        private void button2_Click(object sender, EventArgs e)
        {
            // Botón para cerrar el puerto serie.
            serialPort1.Close();
        }
    }
}

public class VariablesGlobales
{
    public static int botonPulsado = 0;
    public static int MedirVoltajes = 0;
    public static int MedirCorriente = 0;
    public static int MedirTodo = 0;
    public static int BytesLeido = 0;
    public static int NumeroSenosArepresentar = 0;
    public static int bytesPedidos = 0;

    public static List<int> ListaBytesLeido = new List<int>();
    public static List<int> ListaBytesLeido_serialPort = new
List<int>();

    // A parte de leer el dato, nos interesa ir guardandolo en listas
para luego poder acceder a ese dato.
    // Como no sabemos cuantos datos vamos a recibir, usamos listas
dinámicas.
    public static List<double> V_in = new List<double>();
    public static List<double> V_out = new List<double>();
    public static List<double> I_in = new List<double>();
}
public class TramaPerdida
{
    public static List<int> IndiceTramas = new List<int>();
    public static List<int> ValorTramas = new List<int>();
}
public class Trigger
{
    public static int BotonTRIGGER = 0;
    public static double ValorTrigger;
    public static int Trigger_Iin = 0;
    public static int Trigger_Vin = 0;
    public static int Trigger_Vout = 0;

    public static int Flanco_positivo = 0;
    public static int Flanco_negativo = 0;

    public static int ContadorTramaValidaTrigger = 0;

    public static int RepresentarSoloUnaVez = 0;

    // Cada uno de los 7 bytes en los que se va a ir leyendo la trama.
    public static  double V_inByte1, V_inByte2, V_outByte1,
V_outByte2, I_inByte1, I_inByte2;
```

```
    public static  double CRC, I_in, V_in, V_out, suma_TODO,
suma_TODO_mod;

    public static List<double> Lista_Trigger_V_in = new
List<double>();
    public static List<double> Lista_Trigger_V_out = new
List<double>();
    public static List<double> Lista_Trigger_I_in = new
List<double>();
    public static List<int> ListaTrigger = new List<int>();

}
```